

Higher Secondary Course

COMPUTER SCIENCE



Government of Kerala
Department of General Education

State Council of Educational Research and Training (SCERT) Kerala
2026

THE NATIONAL ANTHEM

Jana-gana-mana adhinayaka, jaya he
Bharatha-bhagya-vidhata
Punjab-Sindh-Gujarat-Maratha
Dravida-Utkala-Banga
Vindhya-Himachala-Yamuna-Ganga
Uchchala-Jaladhi-taranga
Tava subha name jage,
Tava subha asisa mage,
Gahe tava jaya gatha
Jana-gana-mangala-dayaka jaya he
Bharatha-bhagya-vidhata
Jaya he, jaya he, jaya he,
Jaya jaya jaya, jaya he.

PLEDGE

India is my country. All Indians are my brothers and sisters.

I love my country, and I am proud of its rich and varied heritage. I shall always strive to be worthy of it.

I shall give my parents, teachers and all elders, respect and treat everyone with courtesy.

To my country and my people, I pledge my devotion.
In their well-being and prosperity alone, lies my happiness.

Prepared by:

State Council of Educational Research and Training (SCERT)

Poojappura, Thiruvananthapuram 695012, Kerala

Website: www.scert.kerala.gov.in, e-mail: scertkerala@gmail.com

Phone: 0471 - 2341883, Fax: 0471 - 2341869

Designed by: SCERT, First Edition: 2026

Printed at: Centre for Advanced Printing and Training (C-apt)

Thiruvananthapuram-695013

© Department of General Education, Government of Kerala

ISBN: 978-93-47213-45-8

Foreword

Dear students,

It gives me great pleasure to present the Plus One Computer Science textbook to you.

Computer Science is a significant area of study in the contemporary world. It helps learners develop logical thinking, problem-solving ability, accuracy and creativity. This textbook has been prepared in a systematic and learner-friendly manner to help you understand the basic concepts of the subject and apply them meaningfully.

The opening section provides the foundation for the study of Computer Science. It introduces concepts such as the evolution of computing, data representation, Boolean algebra, computer system organisation, algorithms and flowcharts. These ideas prepare learners for analytical thinking and decision making.

The major focus of this textbook is problem-solving through Python programming. The book introduces Python in a systematic manner and familiarises learners with control statements, functions and modules, strings, lists, tuples, dictionaries, sets and basic techniques of sorting and searching. This is expected to help students gain confidence in writing programs and solving problems logically.

The textbook also emphasises practice-oriented learning. Activities, examples, programming tasks and exercises have been included to strengthen understanding and improve skill in coding, debugging, testing and analysis. The concluding chapter further highlights the relationship between computers and society, addressing areas such as cyber ethics, intellectual property, assistive technology, free and open source software, e-waste and green computing.

I am confident that this textbook will help you develop interest in the subject and strengthen your ability to think clearly, work systematically and use technology responsibly. I place on record my sincere appreciation to the textbook committee members, subject experts, teachers, reviewers and all others who have contributed to the preparation of this textbook.

I wish all students a fruitful and enriching learning experience.

Dr Jayaprakash R. K.

Director
SCERT Kerala

Textbook Development Team

Chairperson

Prof. (Dr) Murali Krishnan K.
Dept. of Computer Science and Engineering
National Institute of Technology Calicut, Kerala

Experts

Dr Piyush P. Kurur
HoD, Dept. of Computer Science and Engineering
Indian Institute of Technology Palakkad
Kerala

Dr Rubell Marion Lincy G.
HoD, Dept. of Computer Science and Engineering
Indian Institute of Information Technology Kottayam
Kerala

Prof. (Dr) G. Raju
Former HoD, Dept. of Information Technology
Kannur University

Dr Reshma P. K.
Associate Professor
HoD, Dept. of Computer Science
Mahatma Gandhi College, Iritty, Kannur

Members

Asees V.
HSST Computer Science
Govt. HSS Valayam, Kozhikode

Dr Jabir K.
HSST Computer Application
Chakkalakkal HSS, Kozhikode

Joy John
HSST Computer Science
St. Joseph's HSS, Thiruvananthapuram

Dr Nishad A.
Master Trainer KITE
HSST Computer Science

Prasanth P. M.
HSST Computer Science
St. Joseph's Boys' HSS, Kozhikode

Roy John
HSST Computer Science
St. Aloysius HSS, Elthuruth, Thrissur

Language Expert

Prof. (Dr) Deepa L. C.
Chembai Memorial Govt. Music College
Palakkad

Dr Velayudhan Nair T.
PEET Memorial Training College
Mavelikkara, Alappuzha

Artists

Sudheer P. Y.
Chief Animator
C-DIT

Signi Devaraj
Art Teacher (Rtd)
GHSS Neeleswaram, Kozhikkode

Academic Coordinator

Dr Anilkumar A. K., Research Officer, SCERT Kerala



State Council of Educational Research and Training (SCERT)
Vidhya Bhavan, Poojappura, Thiruvananthapuram 695012

<i>Contents</i> _____	<i>Pages</i>
01 Introduction to Computing	07-20
02 Data Representation and Boolean Algebra	21-50
03 Computer System Organisation	51-74
04 Getting Started with Python	75-80
05 Python Fundamentals	81-102
06 Control Statements	103-140
07 Functions and Modules	141-166
08 Strings in Python	167-180
09 Lists	181-206
10 Tuples	207-222
11 Dictionaries and Sets	223-240
12 Sorting and Searching	241-252
13 Computers and Society	253-264

Some icons are used in this textbook for convenience



Additional Info



Let us do

THE CONSTITUTION OF INDIA

PREAMBLE

WE, THE PEOPLE OF INDIA, having solemnly resolved to constitute India into a ¹**[SOVEREIGN SOCIALIST SECULAR DEMOCRATIC REPUBLIC]** and to secure to all its citizens:

JUSTICE, social, economic and political;

LIBERTY of thought, expression, belief, faith and worship;

EQUALITY of status and of opportunity; and to promote among them all

FRATERNITY assuring the dignity of the individual and the ²[unity and integrity of the Nation];

IN OUR CONSTITUENT ASSEMBLY this twenty-sixth day of November, 1949 do **HEREBY ADOPT, ENACT AND GIVE TO OURSELVES THIS CONSTITUTION.**

-
1. Subs. by the Constitution (Forty-second Amendment) Act, 1976, Sec.2, for "Sovereign Democratic Republic" (w.e.f. 3.1.1977)
 2. Subs. by the Constitution (Forty-second Amendment) Act, 1976, Sec.2, for "Unity of the Nation" (w.e.f. 3.1.1977)

Introduction to Computing



Let us pause for a moment and imagine a world untouched by digital technology – without instant calculations, effortless search for information, computer generated entertainment, mobile communication, navigation tools or any AI models to assist our thinking. You can ask your parents or grandparents about their lives in such a world and find out how they learned, worked and stayed connected. Today, computers have opened up a whole new world of wonders and possibilities – we can explore places we might never visit or uncover the hidden rules of nature. This might leave you amazed at how our lives have been transformed, with digital technology becoming such an integral part of daily life. This chapter invites us to unravel that journey and build our own understanding of how computing has evolved to transform the world we experience every day.

This textbook is your gateway into the world of Computer Science. At its heart, Computer Science is all about solving computational problems. Sometimes the problems are simple – like finding the sum of a list of numbers. At other times they are extremely complex – like controlling a spacecraft or securing sensitive information. For a computer to solve a problem, we must break the task into clear, simple steps that the computer can follow. These steps form an algorithm. To communicate our algorithms to a computer, we use programming languages. Python is one such language, which is friendly, powerful, and widely used. When we write the steps of our algorithm in Python, we create a program. The computer then executes the program and solves the problem for us. It must be clear from the onset that computer science is not just a study of computers or programming languages, but more importantly is concerned

with how computers or programming languages can be used to solve problems. The following quote by the famous computer scientist Edsger W. Dijkstra clarifies this point.

“Computer science is no more about computers than astronomy is about telescopes, biology is about microscopes or chemistry is about beakers and test tubes. Science is not about tools. It is about how we use them, and what we find out when we do.”

With this thought, we travel back in time to discover how humans first learned to count, how early machines performed calculations, and how computers evolved generation by generation into the powerful devices we rely on today. We will also navigate through how programming languages developed alongside computers, shaping the way we instruct machines. Finally, we will look into the real-world applications of computers – how they help process data, make decisions, and transform the world around us. So, are you ready? Let’s begin this journey through the evolution of computing – from ancient counting systems to the intelligent digital world we live in now!

1.1 Computing Milestones and Machine Evolution

Long ago, humans faced a simple challenge: how to count and record numbers accurately. To solve this, they invented number systems that helped them make sense of the world around them. But as life grew more complex, counting with fingers and stones was no longer adequate. People began designing clever tools to help them calculate faster. One of the earliest inventions was the Abacus – a simple device that made adding and subtracting easier. Many years later, brilliant minds created mechanical machines like Pascaline, which could perform calculations on their own. These inventions may seem basic to us now, but they were the first steps towards the computers we use today. As time passed, technology advanced rapidly. Machines evolved from slow, bulky devices into fast electronic computers. New ideas, better hardware, and powerful software transformed each generation of computers, making them smaller, smarter, and more capable. From ancient counting methods to the intelligent systems we use today, every milestone has shaped the journey of computing. Now, let us begin this exciting story by looking at how number systems first evolved.

1.1.1 Evolution of Number Systems

Imagine a time when numbers did not exist – people counted using their fingers, stones, or marks on walls. As human activities grew, so did the need for better ways to record and communicate quantities. Different civilizations around the world began inventing their own

number systems. The Egyptians used symbols, the Romans crafted numerals, and the Indians introduced the place-value system and the digit zero, which became a revolutionary idea. Each new system improved how people calculated, traded, and understood the world. Step by step, these contributions formed the strong mathematical foundation on which modern computing stands today. The journey of how numbers evolved over time is briefly shown in Table 1.1.

Table 1.1: Number Systems

Number System	Time Period	Description
Egyptian Number System	3000 BC	- Egyptians used a decimal (base 10) system. - They wrote numbers using picture writing (pictographs), usually from right to left. - Symbols represented powers of ten, such as 1, 10, 100, etc.
Sumerian/ Babylonian Number System	2000 BC	- They used the largest base, which was 60 (sexagesimal system). - This number system was written from left to right. - A space served as a placeholder for zero.
Chinese Number System	1200 BC	- A decimal (base 10) system was used. - Counting rods and boards were employed for calculations.
Greek Number System	500 BC	- Also known as the Ionian number system. - Numbers were written using letters of the Greek alphabet. - It was a non-positional system and did not have a symbol for zero.
Roman Number System	500 BC	- Symbols: I (1), V (5), X (10), L (50), C (100), D (500), M (1000). - Used for counting and recording numbers, non-positional, and did not include zero.
Mayan Number System	300 AD	- A vigesimal (base 20) system. - Included a shell-shaped symbol to represent zero. - The system was positional, which means the value of each digit changed according to its place.
Hindu - Arabic Number System	5 th - 6 th century AD	- Introduced the positional decimal system. - Introduced the symbol 0 for zero, contributed by India. - Forms the basis of the modern number system used worldwide.

1.1.2 Introduction to Computing Machines

The growth of the computer industry started with the need to perform fast calculations. The manual method of computing was slow and prone to errors. Therefore, attempts were made to develop faster calculating tools. Let us first have a look at some early calculating devices.

Abacus

Abacus is considered as one of the earliest tools designed to perform basic arithmetic calculations. It was invented by the Mesopotamians around 3000 BC. It is a rectangular frame holding a series of rods

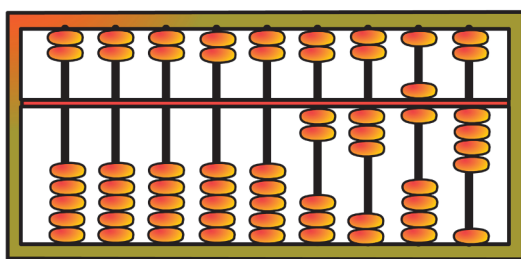


Figure 1.1: Abacus

or wires, on which beads are moved up and down to represent numbers. Addition and multiplication of numbers were done by using the place value of the digits of the numbers and position of beads in an abacus. An abacus model is shown in Figure 1.1. The Chinese further refined the abacus for easier calculations.

1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	1	0	1	2	1
3	3	6	9	1	2	5	1	2	2
4	4	8	1	2	6	2	2	3	3
5	5	0	5	0	2	3	3	4	4
6	6	1	2	8	4	0	6	2	5
7	7	2	4	1	8	3	4	4	6
8	8	3	6	2	4	0	5	6	7
9	9	4	7	3	5	4	3	7	8

Figure 1.2: Napier's Bones

Napier's Bones

The idea of logarithm was developed by John Napier in 1617. He devised a set of numbering rods known as Napier's Bones through which both multiplication and division could be performed (see Figure 1.2). These were numbered rods which could perform the multiplication of any number by a single digit number. There are 10 bones corresponding to the digits 0 – 9 and there is also a special eleventh bone that is used to represent the multiplier. By placing bones corresponding to the multiplier on the left side and the bones corresponding to the digits of the multiplicand on the right, the product of two numbers can be easily obtained.

Pascaline

In 1642, Blaise Pascal, a French mathematician, invented an adding machine (see Figure 1.3), that was made up of gears and wheels and was used for adding numbers quickly. This machine was called

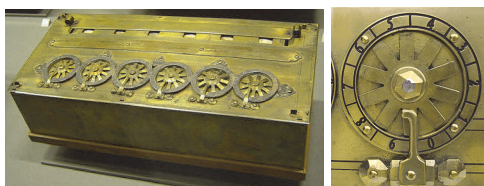


Figure 1.3: Pascaline

Pascaline and was capable of addition and subtraction along with carry-transfer feature. It consisted of various numbered, toothed wheels having unique position values. The addition and subtraction operations were performed by controlled rotation of these wheels.

Leibniz's Calculator

In 1673, Gottfried Wilhelm Leibniz's, a German mathematician extended the capabilities of the adding machine invented by Pascal

to perform multiplication and division as well. This machine was named Leibniz's calculator (see Figure 1.4). The multiplication was done through repeated addition of numbers.

Jacquard's Loom

In order to make different design patterns of the cotton weaving process automatic, Joseph Jacquard devised punched cards and used them to control looms in 1801. These punched cards with different sets of punched holes representing different patterns of designs can be considered as a program to control the looms.

Jacquard introduced a programmable machine as shown in Figure 1.5. The machine could be used to produce other weaving patterns by replacing the current set of punched cards with new sets of punched card instructions. This is a precursor to the programmable computers of the present day. Further, duplicating a set of punched cards enabled the replication of the same design in other looms.

Difference Engine and Analytical Engine

Charles Babbage, an English mathematician, developed a machine called Difference Engine in 1822, which could calculate various mathematical functions. It mechanised a series of calculations on a number of variables to solve a problem. Thereafter, in 1833, he designed the Analytical Engine which later on proved to be the basis of modern computers. This machine could perform all the four arithmetic operations as well as comparison. It included the concept of central processor, memory storage and input-output devices. Even the stored information could be modified. Although the analytical engine was never built at that time, Babbage established the basic principles on which the modern computers work. These two remarkable inventions earned him the title of 'Father of Computers'.

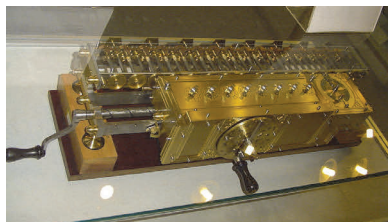


Figure 1.4: Leibniz's calculator

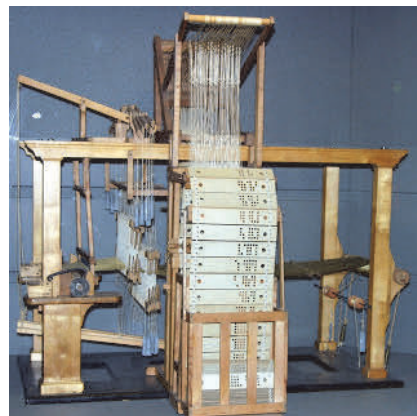


Figure 1.5: Jacquard's loom

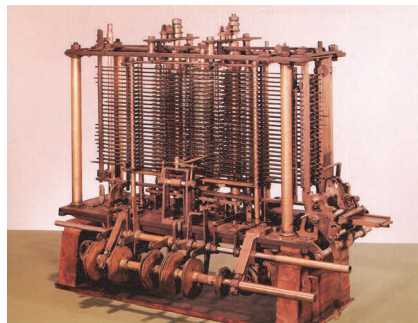
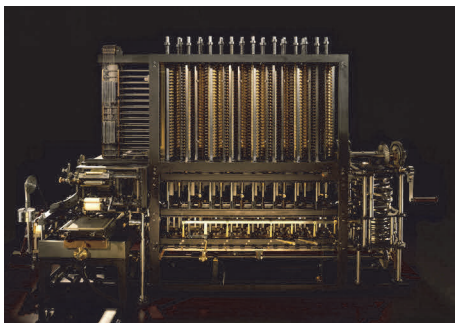
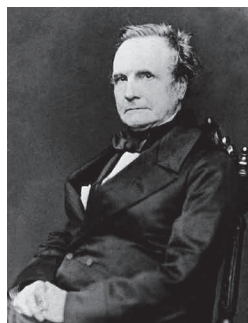


Figure 1.6: Charles Babbage and his inventions



Augusta Ada King, Countess of Lovelace, known as **Ada Lovelace** was a British mathematician and is widely regarded as the first computer programmer. She worked with Charles Babbage on his Analytical Engine and wrote the first algorithm designed for a machine to process. She was the first to realise that computers could do more than just mathematics — they could also create music, art, and graphics when properly programmed. Her ideas laid the foundation for modern computing.



Mark I

Prof. Howard Aiken, in collaboration with IBM, constructed an electromechanical computer named Mark I in 1944. It could multiply two 10-digit numbers in 5 seconds. This machine was based on the concept of Babbage's Analytical Engine and was the first operational general-purpose computer which could execute preprogrammed instructions automatically without any human intervention. Mark I played an important role in computer history by bridging the age of mechanical calculators and the emergence of automatic computers.

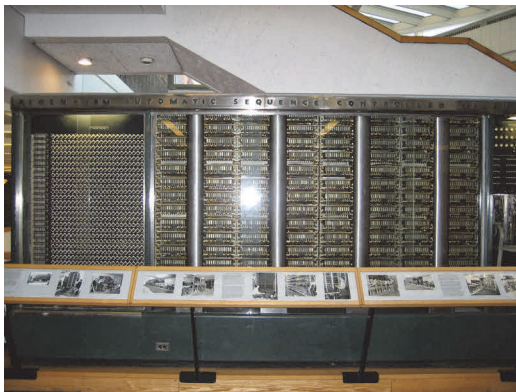


Figure 1.7: Mark I

Stored Program Computer

In 1945, Dr. John von Neumann proposed the concept of a stored program computer. As per this concept, the program and data could be stored in the same memory unit. The basic architecture of the von Neumann computer is shown in Figure 1.8.

This was a major shift from the earlier machines where instructions were fixed in hardware wiring or punched cards. All modern computers are based on the stored program concept.

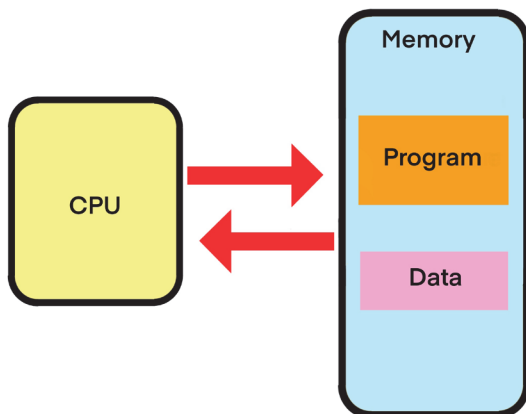


Figure 1.8: Stored program computer



Let us do

- Which among the following machines were programmable? Justify your answer.
a) Jacquard's loom b) Pascaline c) Analytical engine d) Leibnitz's calculator
- Arrange the following number systems in the correct chronological order of their origin:
Roman, Egyptian, Hindu-Arabic, Sumerian/Babylonian, Mayan.
- Arrange the following number systems in the correct chronological order of their base : Egyptian, Mayan, Sumerian/Babylonian

1.1.3 Generations of Computers

Table 1.2: Generations of computers

Generation	Standard Dates (Approximate)	Technology
First	1940 – 1956	Vacuum Tubes
Second	1956 – 1963	Transistors
Third	1964 – 1971	Integrated Circuits (ICs)
Fourth	1971 – Present	Microprocessors
Fifth	1982 – Present	Artificial Intelligence (AI)

Technological development is the primary factor driving the growth of the computer industry. Computers are grouped into distinct generations, each defined by a major advancement in hardware or software that significantly improves the overall system performance.

These shifts resulted in computers becoming progressively smaller, faster, affordable and more reliable. Based on the various stages of development, computers can be grouped into different generations as shown in Table 1.2.

First Generation (1940 – 1956)

The first generation computers used thousands of vacuum tubes and were the fastest calculating devices of their time. A vacuum tube, as shown in Figure 1.9, is a delicate glass device that can control and amplify electronic signals. These computers were bulky in size, consumed large amount of electricity and generated excessive heat. ENIAC (Electronic Numerical Integrator and Computer), EDVAC (Electronic Discrete Variable Automatic Computer) and UNIVAC I (Universal Automatic Computer I) were the popular computers in this generation.



Figure 1.9: Vacuum tube

Second Generation (1956 – 1963)

The transistor is the main technological development in this generation. It was invented in 1947 at Bell Labs by a team of physicists including John Bardeen, Walter H. Brattain and William Shockley. A transistor (see Figure 1.10) is a solid-state semiconductor device that could replace vacuum tubes. They were much faster, smaller in size, required considerably less power, and generated far less heat than vacuum tubes. Further, magnetic core memories were developed during this generation, making long term data storage possible in a cost-effective way. All these new developments made computers more powerful and reliable. Further, these technological advancements led to advances in software development. Operating systems such as GMOS for IBM machines and programming languages like FORTRAN, COBOL, Algol etc. were developed. Commercial applications of computers increased, and they were widely used in business and industries for applications such as payroll management, employee data storage, inventory control etc. IBM 1401 and IBM 1620 were popular computers of this generation.

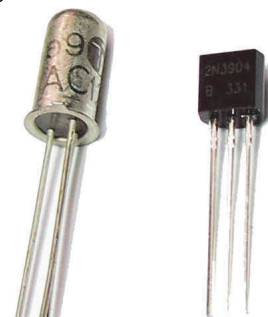


Figure 1.10: Transistors

Third Generation (1964 – 1971)

The third generation of computers was characterised by the use of integrated circuits (ICs). An IC is a small silicon chip or wafer made up of extremely purified silicon crystals that could contain numerous transistors, capacitors, resistors and other elements of an electronic circuit (see Figure 1.11). The memory capacity also increased. Various mainframe computers and minicomputers were developed during this generation. Even operating systems with multitasking and multiprogramming features such as variants of OS/360 by IBM were developed. Since ICs made the computers faster, highly reliable and relatively inexpensive, they were found in areas of education, small businesses and offices along with industrial and business applications. IBM 360, IBM 370 and PDP-8, PDP-11 are some popular third generation computers.

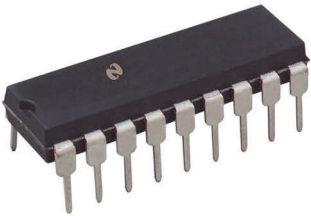


Figure 1.11: IC chip

Fourth Generation (1971 – Present)

The microprocessor (see Figure 1.12) is the key technology that defined the fourth generation of computers. In this generation, Large Scale Integration (LSI) and Very Large Scale Integration (VLSI) technologies, where thousands of transistors integrated on a single IC were used. This led to the development of microprocessors, which were affordable in cost and size and could be designed for personal use. They had faster accessing and processing speeds. In addition to this, the increased memory capacity further made the computers more powerful. More complex and efficient operating systems such



Figure 1.12: Microprocessor

as Unix were developed for these computers. The use of computer networks and the Internet started during the fourth generation of computers. Personal computers (PCs) were introduced during this generation, making computers available for individual and business use. The first microprocessor, Intel 4004, was developed in 1971 marking the beginning of the fourth generation. Most of the computers which we use today are fourth generation computers.

Fifth Generation (1982 – Present)

The fifth generation of computers is known as the generation of Artificial Intelligence (AI) because it focuses on developing computers that can learn, think and make decisions, imitating the cognitive functions of the human brain. The fifth generation of computers is not just about AI, it also involves several advanced technologies that make computers smarter, faster, and more connected. This generation combines machine learning, Natural Language Processing



Figure 1.13: AI technology

(NLP), robotics, parallel processing, cloud computing and quantum computing to create intelligent, high-performance and highly connected systems. The fifth generation of computers is generally considered to have begun in 1982, and it continues to evolve.

1.1.4 Evolution of Programming Languages

As computers evolved through different generations, the languages used to program them also changed significantly. In the early stages of computing, programming was closely tied to the hardware and required detailed knowledge of machine operations. Over time, programming languages were developed to reduce complexity, improve efficiency, and make computers easier to use. This gradual development from low-level to high-level languages is known as the evolution of programming languages.

The earliest form of programming was machine language, which consists of instructions written in binary digits, that is, 0s and 1s. These instructions are directly understood by the computer's central processing unit. Although machine language allows fast execution, it is very difficult for programmers to write, understand, and debug. Moreover, machine language programs are machine dependent, which means a program written for one computer cannot be used on another.

To overcome the difficulties of machine language, assembly language was introduced. Assembly language uses mnemonic codes or symbolic instructions such as ADD, SUB, and MOV instead of binary numbers. This made programming easier and reduced errors. Assembly language programs are machine dependent, which means the same program cannot run on different types of computers without modification. Due to these limitations, there arose a need for more user-friendly and machine-independent programming languages, leading to the development of high level languages.

High level languages use statements that resemble natural English language, making them easy to learn and use. In 1950s John Mauchly proposed a simple high level language called Short Code. Short Code consisted of mathematical expressions which were interpreted step by step. Later a miniature programming language, called Autocode was developed by Alick Glennie in the early 1950s. However, the first widely used high level language was FORTRAN (1957), developed by a team lead by John Backus for IBM. Subsequently popular programming languages such as LISP (1958), COBOL (1959), BASIC (1964) Pascal (1970), Prolog (1972), C (1972), C++ (1983), Python (1991), JavaScript (1995), Java (1995) etc. were developed.



Along with programming languages, various programming methodologies known as paradigms of programming also evolved. These methodologies differ in the way they model a computational problem and modularise the code. Some common programming paradigms are imperative (procedural) programming, object-oriented programming, functional programming and logic programming. Among the languages listed above, FORTRAN, COBOL, Pascal and C are imperative programming languages. C++, Java, Python and JavaScript are object-oriented. LISP is a functional language whereas Prolog is a logic programming language.



Let us do

1. List down two programming languages that were not mentioned in the text book, such that at least one of them was invented after 1990. For each language, find out who invented the language and which programming paradigms are supported?
2. Correct all five pairs by matching the computer generation with its correct key technology.

(First Generation, Transistors)	(Second Generation, ICs)
(Third Generation, AI)	(Fourth Generation, Vacuum Tubes)
(Fifth Generation, Microprocessors)	

1.2 How are computers used in practice?

A functional computer system requires both the physical equipment, called the hardware, and certain built in programs that accompany the hardware before delivery to the user. The term software broadly refers to programs written for execution on a computer. We will learn more about hardware and software in Chapter 3.

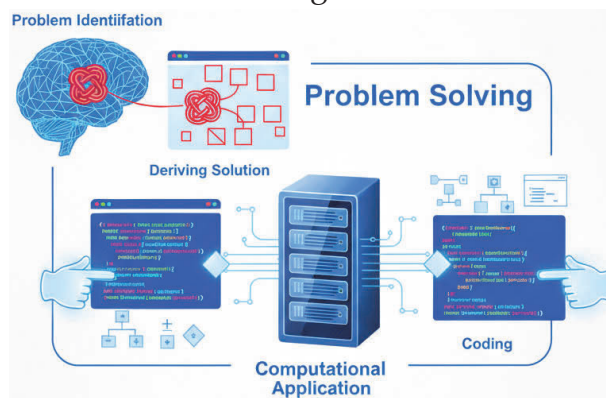
Computers can be used for a variety of purposes. The mobile devices which we use in daily life are small computer systems. Similarly, laptops, notebook computers and normal desktop machines are examples of computer systems. Some of the main tasks for which computers are used include the following.

- **Data Storage and Processing**

Computers can be used to store massive amounts of data. For instance, the Aadhar database contains information about crores of people in our country, and the data is stored across a complex network of computers. Word processing, payrolls, accounting and other statistical applications can be quickly performed on large amounts of data by using computers. Normally, for data processing, specialised programs designed and developed by experts are used. Examples of such software includes Database Management Systems, Word processors, spreadsheets and so on. We will discuss data processing in the next section.

• Problem Solving

We can design our own programs for solving computational problems. To achieve this, we must be able to design an algorithm for a given problem and express the algorithm in a programming language such as Python so that it can be executed on a computer. Programming (or coding) is a broad term associated with this activity. We will learn how to design programs in the Python programming language later in this book.



To make our task of program writing easier, several specialised software such as compilers/ interpreters and operating systems are available along with a computer system. We will be using a Python interpreter that is supplied along with the Linux Operating System during your laboratory sessions.

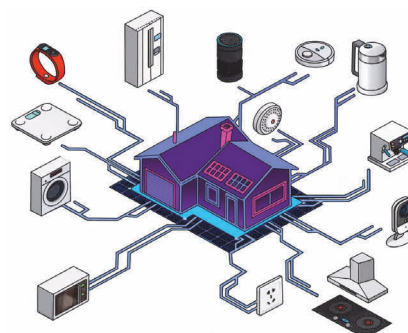
• Networking and Communication

We can send and receive data from one computer system to another across communication networks. Computer networking is a broad term that describes the devices and technologies used to handle communication. It allows computers to connect and share information, files and hardware resources. The Internet, the world's largest network, enables fast communication through emails, video calls, social media, and various online services.

Networking also supports a wide range of entertainment activities. Using computers and the internet, people can watch movies, listen to music, play online games, stream videos, and enjoy social media platforms. These services make entertainment more interactive, enjoyable, and easily accessible from anywhere. Thus, networking not only improves communication but also enhances entertainment in our daily life.

• Intelligent Decision Making

A modern trend in the use of computers is to run smart programs that can understand data and make decisions like humans. For example, computers can help doctors diagnose diseases, play chess better than people, and even drive cars on their own. These developments are possible because of improvements in communication networks, computer hardware, and software. Techniques like machine learning and artificial intelligence are used in many such applications. We will learn some basic ideas of machine learning next year.



Another important area is the Internet of Things (IoT). It connects everyday devices like home appliances, cars and wearable gadgets, to the internet. These devices can collect information and make smart actions, such as a smartwatch tracking your health or a smart bulb turning on automatically. IoT makes our surroundings smarter and more convenient.

In this textbook, we mainly focus on problem solving. We shall look a little deeper into the principles underlying hardware and software design as well.

1.3 Computers for Data Processing

As technology progresses, the techniques used for collecting, processing, and presenting data have transformed significantly. Earlier systems depended heavily on manual data entry, whereas modern computer systems receive information automatically through web interfaces, smart sensors, AI-powered devices, and even from interactive gaming platforms. Whether it involves a website that handles user input, an IoT network tracking environmental conditions, or a game reacting to live player actions, today's systems process data with remarkable speed, intelligence, and precision. Whatever be the data and the method of collection, the data processing mainly involves the following activities.

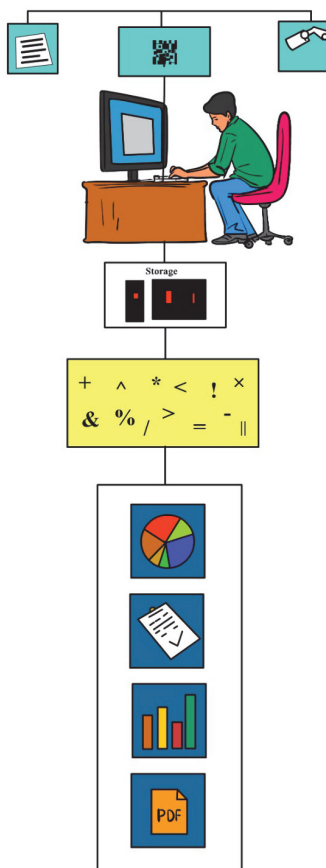
Input Format Design

Input format design involves deciding in advance how data should be structured before it is collected. For this, suitable formats are designed and this step ensures that the data is uniform, accurate, and suitable for further processing. Here are some situations, where prescribed formats are used for data collection.

- A billing/invoicing system defining the format for a customer order record.
- A website registration form for an admission system capturing the required personal and academic details.
- IoT sensor configuration deciding measurement units and frequency.
- AI cameras setting formats for storing images, video frames, and recognition results, so they can be analysed accurately later.
- Game UI layout defining how players provide commands.

Data Input

This step captures the data from various input devices or sources. Data input can be manual, automated, or sensor-driven, depending on the type of system. The following are some of the tools or facilities used to input data.



- A billing/invoicing system scanning product barcodes at a point of sale (POS) system to capture product codes and prices.
- Web forms collecting the details of students.
- IoT sensors sending temperature, motion and air-quality readings.
- AI cameras feeding video frames for recognition.
- Controllers and Virtual Reality (VR) devices capturing player movement in games.

Data Storage

Before or after processing, data must be stored safely so that it can be reused whenever needed. Storage devices such as hard disks, Solid State Drives (SSDs), cloud storage and databases help preserve data for short-term and long-term use. Proper data storage ensures data security, quick retrieval, backup, and continuous availability for future processing or decision-making.

- Billing records are stored for accounting and customer service.
- Online application details are stored in databases.
- IoT readings are saved for analysis over time.
- AI cameras store images and recognition results for security checks.
- Games save player progress and settings.

Data Processing

Data processing transforms raw input into meaningful information using algorithms, databases and intelligent models. This stage involves operations such as validation, computation, comparison and decision making. Some of the processes involved in the different scenarios are given below.

- Billing/Invoicing computation: Multiplying the quantity by the unit price for each item and summing them up to calculate the Total Bill Amount.
- Web servers verifying the inputs, computing and preparing admission list.
- Industrial IoT systems analysing sensor values to detect anomalies.
- AI systems identifying faces, objects or activities from video.
- Game engines performing movement calculations, animation, and AI-driven actions.

Output Generation and Visualisation

This step presents the processed results to the user in a clear and useful form. Outputs may be displayed as text, graphics, sound, alerts, dashboards, or interactive visualisations. Visualisation helps users to interpret patterns, trends, and insights through charts, graphs, maps and other graphical representations. Let us have a look at some of the outputs obtained after data processing in different applications.

- Billing/Invoicing system printing or generating a PDF of the final invoice or bill for the customer.
- Web portals showing reports, charts, or confirmation messages.
- Smart sensor dashboards providing alerts and real-time graphs.
- AI camera interfaces reporting detected objects and analytics.
- Game screens showing animations, scores, maps, and replays.



Let us do

1. Many modern programming languages support multiple paradigms. Find out which are the paradigms supported by the following modern programming languages a) Scala b) Rust
2. Prepare a chart showing a brief description of two major IT companies in the world, two major IT companies founded in India, and two IT companies that originated in Kerala.
3. Read about the Turing award by the Association of Computing Machinery (ACM). Prepare a short presentation of the contributions of any two Turing award winners - one who won the award before 1990 and one who won it after 2015. In each case, present a brief summary of the major contributions of the winner to computer science.

In this chapter, we have traced the early developments that shaped the world of computing, beginning with ancient number systems which formed the foundation for counting, calculation, and eventually computational thinking. We have also explored the conceptualising of early computing machines and their gradual evolution across generations, along with the parallel growth of programming languages. The chapter also highlights how computers are used in practice for data processing, through real life examples. This background prepares us to move from what computers are and how they evolved to how computers actually work internally. In the next chapter, we will look into number systems in detail, internal data representation, and Boolean algebra – the concepts that form the logical and mathematical basis of all computer operations and are essential for deeper learning in Computer Science.

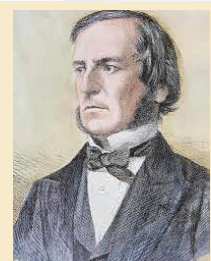
Data Representation and Boolean Algebra

After understanding the evolution of computing and the role of computers in modern applications, it is important to explore how computers actually store, represent, and process information internally. This chapter introduces the fundamental concepts that enable computers to work with data. It explains how different types of data such as numbers, text, images, audio, and video are represented using number systems and binary form. It also explains how this data is arranged and stored inside a computer's memory. The method of storing and expressing data in binary form is known as **data representation**. The chapter also discusses internal data representation techniques that allow computers to handle both numerical and non-numerical data accurately and efficiently.

In addition, the chapter introduces Boolean algebra, which deals with logical values and operations such as AND, OR, and NOT. These logical principles, developed by George Boole, form the basis of digital circuits, control flow in programs, and decision-making processes within a computer. Together, these concepts help you understand how real-world information is converted into binary form, logically processed, and transformed into meaningful computational results, providing a strong foundation for further study in Computer Science.

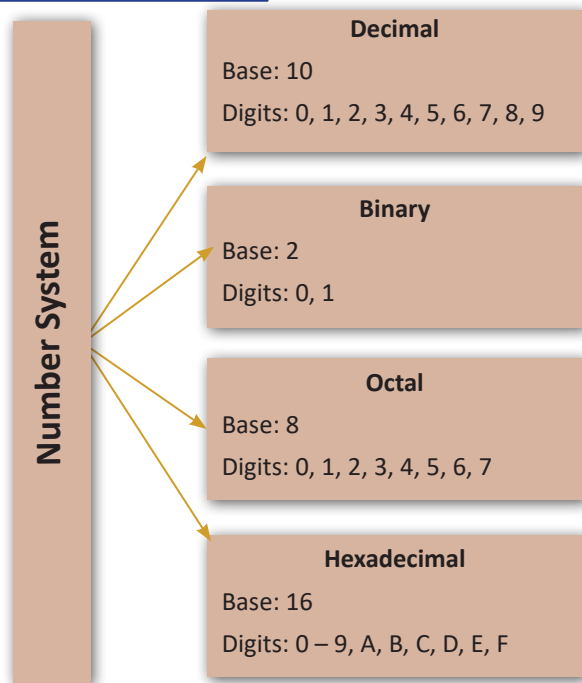


George Boole (1815–1864) was a self-taught English mathematician who developed Boolean Algebra, which later became the foundation of computer logic and digital circuits. Boole did not attend college, yet he became a professor at Queen's College, Cork (Ireland) because of his deep knowledge and passion for mathematics. Although he lived long before computers were invented, his logical system of True (1) and False (0) values became the core principle behind modern computers and programming. Today, Boolean logic is used in search engines, databases, and digital electronics.



2.1 Number System

We know that in a positional number system, the value of each digit of a number depends on its face value (the digit itself) and its place value (its position in the number). Such systems are commonly known as number systems. Each number system uses a fixed set of unique symbols or characters called digits, and the total number of these digits is known as the base or radix of the system. The place value of a digit is expressed as a power of the



2.1: Classification Figure of Number System

base. The positional number system helps in representing large numbers efficiently using only a small set of digits. In this chapter, we will explore different types of number systems and understand their importance in computer operations.

Computers use the binary number system for its working, while humans use decimal and other number systems to easily understand and interact with computers. The four main number systems used are shown in Figure 2.1. The binary number system is the most important in computers, as all data and instructions are represented and processed internally using only two digits: 0 and 1. The octal and hexadecimal systems are often used as shorthand forms of binary numbers to make large binary values easier to read

and understand. Although the decimal system is not used inside computers, it is important for showing results and for interaction with users because it is the system we use in our daily life. We restrict our discussion to binary, decimal and hexadecimal number systems in this chapter.

2.1.1 Decimal Number System

The decimal number system is the one we commonly use in our daily life. It is called the base-10 system because it uses the ten digits, from 0 to 9. In this system, each number is represented by two factors – the digit itself (symbol value) and its position in the number (positional value based on base-10). Consider the number 5398. The following table shows the position values (place values) of the digits and the mode of computation of the number.

Digits	5	3	9	8
Place values	10^3	10^2	10^1	10^0

Constitution of the number

$$\begin{aligned}
 &= 5 \times 10^3 + 3 \times 10^2 + 9 \times 10^1 + 8 \times 10^0 \\
 &= 5 \times 1000 + 3 \times 100 + 9 \times 10 + 8 \times 1 \\
 &= 5000 + 300 + 90 + 8 \\
 &= 5398
 \end{aligned}$$

In the above example, the digit 5 has the highest place value ($10^3 = 1000$) and 8 has the lowest place value ($10^0 = 1$). The digit with the highest place value is known as the Most Significant Digit (MSD) and the digit with the lowest place value as the Least Significant Digit (LSD). It is obvious that the left most digit of a number is the MSD and the rightmost digit the LSD.

Let us consider another number 385.76, which has both an integer part and a fractional part. The following table shows the digits and their corresponding place values.

Digits	3	8	5	.	7	6
Place values	10^2	10^1	10^0	+	10^{-1}	10^{-2}

Constitution of the number

$$\begin{aligned}
 &= 3 \times 10^2 + 8 \times 10^1 + 5 \times 10^0 + 7 \times 10^{-1} + 6 \times 10^{-2} \\
 &= 3 \times 100 + 8 \times 10 + 5 \times 1 + 7 \times \frac{1}{10} + 6 \times \frac{1}{100} \\
 &= 300 + 80 + 5 + 0.7 + .06 \\
 &= 385.76
 \end{aligned}$$

2.1.2 Binary Number System

Computer processors mainly rely on Integrated Circuits (ICs) which consist of many transistors that work based on the electronic signals they receive – either low or high. The high (ON) state of a transistor is represented by 1, and the low (OFF) state is represented by 0. These two digits, 0 and 1, form the binary number system (bi means two). It is also called the base-2 system because it uses only two digits. Each digit in a binary number is called a bit which stands for binary digit. Some examples of binary numbers are 10111, 110.01, 110011. Since these numbers are also there in the decimal number system, it is mandatory to use the base as the subscript of these numbers to distinguish them from those in the other systems. Thus it is clear that the numbers $(10111)_2$, $(110.01)_2$, $(110011)_2$ belong to the binary number system. This notation should be strictly followed in all number systems except the decimal number system. Let us consider the number $(11010)_2$ to illustrate the concept of place values and the value of the number.

Digits	1	1	0	1	0
Place values	2^4	2^3	2^2	2^1	2^0

Constitution of the number

$$\begin{aligned}
 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\
 &= 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 \\
 &= 16 + 8 + 0 + 2 + 0 \\
 &= 26
 \end{aligned}$$

An important point to note is that the number 26 is a decimal number, which is equivalent to the binary number $(11010)_2$. In the case of a binary number, the leftmost bit is known as Most Significant Bit (MSB) and the rightmost bit as the Least Significant Bit (LSB).

2.1.3 Hexadecimal Number System

The hexadecimal number system consists of 16 unique symbols (0–9 and A–F) and is known as the base-16 system. This system provides a compact way to represent binary numbers. In the hexadecimal

Table 2.1: Decimal and Binary equivalents of Hexadecimal numbers

Hexadecimal Number	Decimal Value	4-bit Binary Number
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

system, each digit corresponds to a group of 4 binary digits since 4 bits ($2^4 = 16$) are enough to represent 16 distinct symbols. Table 2.1 shows the decimal and the binary equivalents of the 16 symbols used in the hexadecimal number system. The decimal numbers 10 to 15 are represented by the letters A to F. Examples of hexadecimal numbers include $(59.43)_{16}$, $(A2B)_{16}$ and $(619B.0A)_{16}$. Let us have a look at the concept of place values and decimal equivalency using the number $(10A)_{16}$.

Digits	1	0	A
Place values	16^2	16^1	16^0

Constitution of the number

$$\begin{aligned}
 &= 1 \times 16^2 + 0 \times 16^1 + A \times 16^0 \\
 &= 1 \times 256 + 0 \times 16 + 10 \times 1 \\
 &= 256 + 0 + 10 \\
 &= 266
 \end{aligned}$$



Hexadecimal numbers are commonly used in memory addresses, colour codes in web design, and machine-level programming. Usually, the size of a memory address ranges from 16 bit to 64 bit. Consider a 16-bit memory address 1100000011110001. In the hexadecimal form, it is represented as C0F1, which is easier to read and remember. While describing colours in web pages, the primary colours Red, Green, Blue (RGB) are used. In most colour maps, each colour is selected from a palette of about 16 million colours. Hence, 24 bits are used to represent each colour, with 8 bits each for the Red, Green and Blue components. Since a 24-bit binary colour code is difficult to remember, colours are represented in hexadecimal form for compact and convenient notation.

2.2 Conversion Between Number Systems

In the previous section, you might have observed that both binary and hexadecimal numbers have corresponding decimal values. Therefore, it is obvious that a number in one number system has equivalent representations in the other number systems. Let us now learn how to convert numbers from one system to another.

2.2.1 Decimal to Binary Conversion

The method of converting a decimal number to any other number system is repeated division by the destination base. Since the base of the binary system is 2, follow the steps given below for the conversion.

1. To convert a decimal number to the binary system, divide the number repeatedly by 2.
2. Continue dividing until the quotient becomes 0.

3. Note down the remainder after each division.
4. Finally, write the remainders in the reverse order to get the binary number.

Example: To find the binary equivalent of 29.

2	29	
2	14	1
2	7	0
2	3	1
2	1	1
	0	1



Let us do

Find the binary equivalents of the following decimal numbers.

(a) 39

(b) 98

(c) 127

$$29 = (11101)_2$$

2.2.2 Decimal to Hexadecimal Conversion

Since the target base is the hexadecimal system, the division process is performed by 16 as specified in the following steps.

1. To convert a decimal number to a hexadecimal, divide the number repeatedly by 16.
2. Continue dividing until the quotient becomes 0.
3. Note down the remainder after each division.
4. Substitute the remainders 10 to 15 with the respective letters from A to F.
5. Finally, write the remainders in the reverse order to get the hexadecimal number.

Example: To find the hexadecimal equivalent of 381.

16	381	
16	23	13 (D)
16	1	7
	0	1



Let us do

Find the hexadecimal equivalents of the following decimal numbers.

(a) 155

(b) 2584

(c) 495

$$381 = (17D)_{16}$$

2.2.3 Binary to Decimal Conversion

In section 2.1.2, we illustrated how the decimal value of a binary number is obtained. The same procedure is applied to convert a binary number into the decimal system. The following steps are to be carried out for this.

1. Write the binary number clearly and identify each bit.
2. Assign place values to each bit starting from the rightmost digit. The rightmost bit has a place value of 2^0 , the next is 2^1 , then 2^2 , and so on.
3. Multiply each bit by its corresponding power of 2.

4. Add all the results obtained from the multiplications.
5. The final sum is the decimal (base-10) equivalent of the binary number.

Example: To find the decimal equivalent of $(101101)_2$.

Digits	1	0	1	1	0	1
Place values	2^5	2^4	2^3	2^2	2^1	2^0

Decimal equivalent

$$\begin{aligned}
 &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
 &= 1 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 \\
 &= 32 + 0 + 8 + 4 + 0 + 1 \\
 &= 45
 \end{aligned}$$

$$(101101)_2 = 45$$



Let us do

Find the decimal equivalents of the following binary numbers.

(a) $(101010)_2$

(b) $(1101110)_2$

(c) $(10001011)_2$

2.2.4 Binary to Hexadecimal Conversion

From Table 2.1, it is evident that a hexadecimal digit can be constituted by a maximum of 4 bits. Hence, for any binary number, its hexadecimal equivalent can be computed by performing the following steps.

1. Divide the binary number into groups of 4 bits, starting from the right.
2. If the leftmost group has less than 4 bits, add zeros on the left.
3. Replace each 4-bit group with its corresponding hexadecimal symbol.
4. Combine all hexadecimal symbols to get the final number.

Example: To find the hexadecimal equivalent of $(1101001011)_2$.

Number in 4-bit groups	0011	0100	1011
Hexadecimal symbol	3	4	B

$$(1101001011)_2 = (34B)_{16}$$



Let us do

Find the hexadecimal equivalents of the following binary numbers.

(a) $(11010101)_2$

(b) $(1100000010)_2$

(c) $(1001010100010101)_2$

2.2.5 Hexadecimal to Binary Conversion

This conversion is the reverse process of the previous one. As each hexadecimal symbol is represented by a group of 4 bits, the following steps are used to convert a hexadecimal number into its binary form.

1. Write down the hexadecimal number.
2. Replace each hexadecimal digit with its 4-bit binary equivalent (refer to Table 2.1).
3. Join all the 4-bit groups together to get the complete binary number.

Example: To find the binary equivalent of $(7B0D)_{16}$.

Hexadecimal symbol	7	B	0	D
4-bit binary equivalents	0111	1011	0000	1101

$$(7B0D)_{16} = (0111101100001101)_2$$



Let us do

Find the binary equivalents of the following hexadecimal numbers.

- (a) $(2AB)_{16}$ (b) $(90F5)_{16}$ (c) $(FF57AB)_{16}$



The hexadecimal number system is widely used in computing because it makes working with binary values easier. A single hexadecimal digit represents four binary bits and hence long binary numbers can be written in a shorter and more readable form. Thus, hexadecimal acts as a convenient shortcut between binary used by computers and decimal used by humans.

2.3 Binary Addition and Subtraction

After number conversions, the next important operations in the binary system are binary addition and binary subtraction. Understanding these operations is important because all arithmetic calculations inside a computer are carried out using binary numbers. These operations are similar to decimal addition and subtraction, but are performed using only two digits – 0 and 1. In binary addition, specific rules are followed to add two binary digits as shown in Table 2.2. The following procedure is to be carried out to perform binary addition.

1. Write the two binary numbers in column form, aligning the bits of both the numbers according to their place value.
2. Starting from the right (LSB), add the bits following the rules depicted in Table 2.2.
3. Write the result of the addition of the corresponding bits in the current column.
4. If there is a carry 1 (i.e., $1 + 1 = 10$) in the result, add the carry to the pair of bits at the immediate left column.
5. Repeat the same process for every bit till the leftmost side.
6. If a carry remains at the end, write it as the MSB.

Table 2.2: Binary addition rules

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Note that a carry bit 1 is created only when two 1s are added. If three 1s are added (i.e., $1 + 1 + 1$), then the sum bit is 1 with a carry bit 1.

Example: Add the numbers
 $(1001)_2$ and $(101)_2$.

$$\begin{array}{r} 1001 \\ 101 \\ \hline 1110 \end{array}$$

Example: Add the numbers
 $(101101)_2$ and $(10011)_2$.

$$\begin{array}{r} 101101 \\ 10011 \\ \hline 1000000 \end{array}$$

As in the case of binary addition, specific rules are followed in binary subtraction as shown in Table 2.3. Note that a borrowing rule is applied when subtracting a larger bit from a smaller one. That is, when 1 is subtracted from 0, the difference is 1, but 1 is borrowed from the immediate left bit of the first number. The procedure to perform binary subtraction is given below.

Table 2.3: Binary subtraction rules

A	B	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

1. Write the binary numbers in column form, aligning both numbers bit by bit with the minuend (number from which we subtract) on the top.
2. Start subtracting from the rightmost bit (LSB).
3. Apply the basic binary subtraction rules as shown in Table 2.3.
4. If the top bit is 0 and the bottom bit is 1, you need to borrow from the next higher bit (Rule 2 in Table 2.3).
5. When 1 is borrowed from the next higher bit, its immediate right bit 0 becomes 10 (2 in decimal). After the borrow, it becomes 1.
6. Continue subtracting bit by bit until the leftmost side.

Example: Subtract $(1001)_2$ from $(1101)_2$.

$$\begin{array}{r} 1101 \\ 1001 \\ \hline 100 \end{array}$$

Example: Subtract $(10111)_2$ from $(101000)_2$.

$$\begin{array}{r} 101000 \\ 10111 \\ \hline 10001 \end{array}$$



Let us do

1. Carry out the following binary additions.
(a) $(10011)_2 + (11011)_2$ (b) $(100011)_2 + (1110)_2$ (c) $(11011011)_2 + (1101111)_2$
2. Perform the following binary subtractions.
(a) $(10011)_2 - (1101)_2$ (b) $(100001)_2 - (11111)_2$ (c) $(10101010)_2 - (10101)_2$

2.4 Data Representation

In a computer system, all types of data such as numbers, text, images, audio and video are represented in binary form using 0s and 1s. This process of expressing different kinds of data in a format that a computer can understand is called data representation. In this section, we discuss how computers represent these data internally.

2.4.1 Representation of Integers

Integers include both positive and negative whole numbers. In computer systems, integers are represented in binary form using a fixed number of bits. Since a binary digit can represent only two values, special methods are used to indicate the sign (positive or negative) of an integer. The common methods for representing signed integers are

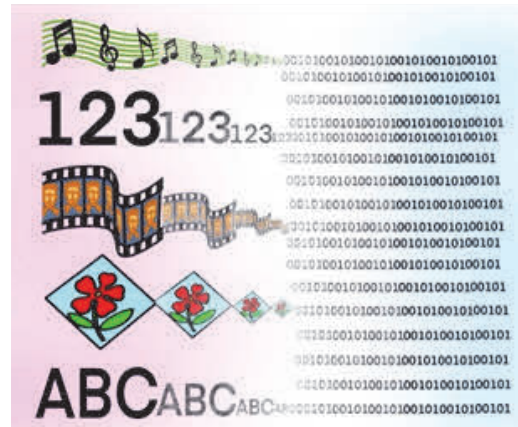
- i. Sign and magnitude representation
- ii. 1's complement representation
- iii. 2's complement representation

For simplicity, we will use an 8-bit word length to represent an integer. A word is a fixed-size group of bits processed by the computer's processor. The total number of bits in a word is known as its word length. The word length depends on the architecture of computer and common word lengths include 8, 16, 32, and 64 bits.

(i) Sign and Magnitude Representation

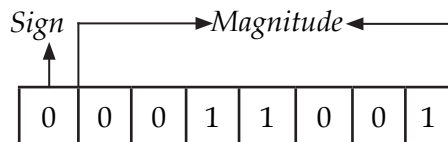
The sign and magnitude method is a simple way to represent positive and negative numbers in binary form. In this method, the leftmost bit (MSB) is called the sign bit, which is 0 for positive numbers and 1 for negative numbers. The remaining bits show the magnitude or actual value of the number. This method helps us easily identify whether a number is positive or negative by looking at the sign bit. The steps to represent an integer in sign and magnitude form are given below.

1. Convert the magnitude of the given decimal number into binary form using the required number of bits (Since we follow an 8-bit word length, express the binary number in 7 bits).
2. Determine the sign of the number. Set the MSB as 0 for a positive number and 1 for a negative number.
3. Combine the sign bit and magnitude bits.
4. The resulting binary number is the sign and magnitude representation of the integer.



Example: Represent +25 in sign and magnitude form.

Binary equivalent of the magnitude (25) = 11001



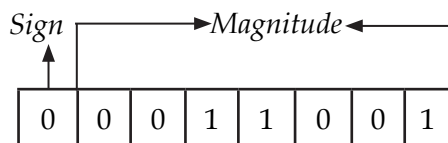
Binary number of 25 in 7 bits = 0011001

Sign bit (MSB) of the number = 0

Thus, +25 in Sign & Magnitude form = $(00011001)_2$

Example: Represent -109 in sign and magnitude form.

Binary equivalent of the magnitude (109) = 1101101



Binary number of 109 in 7 bits = 1101101

Sign bit (MSB) of the number = 1

Thus, -109 in Sign & Magnitude form = $(11101101)_2$

(ii) 1's Complement Representation

The 1's complement method is another way to represent positive and negative numbers in binary form. In this method, positive numbers are written in their normal binary form. To represent a negative number, we invert all the bits of its positive binary form – that means changing every 0 to 1 and every 1 to 0. This method helps computers perform subtraction easily using addition. The following are the steps to represent an integer in 1's complement form.

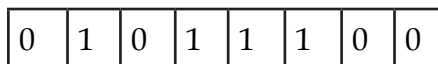
1. Find the binary equivalent of the absolute value (magnitude) of the given number and express it using the required word length.
2. If the decimal number is positive, write the binary form obtained in step 1 as the 1's complement of the given number.
3. If the decimal number is negative, invert all the bits of this binary number – change every 0 to 1 and every 1 to 0.
4. The resulting binary number is the 1's complement representation of the negative integer.

Example: Represent +92 in 1's complement form.

Binary equivalent of the magnitude (92) = 1011100

Binary number of 92 in 8 bits = 01011100

Thus, +92 in 1's complement form = $(01011100)_2$

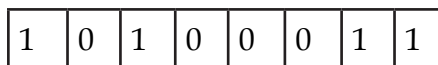


Example: Represent -92 in 1's complement form.

Binary equivalent of the magnitude (92) = 1011100

Binary number of 92 in 8 bits = 01011100

Thus, -92 in 1's complement form = $(10100011)_2$



In Sign & Magnitude and 1's Complement forms, 8-bit word can represent $2^8 - 1 = 255$ numbers from -127 to +127. Similarly, 16-bit word can represent $2^{16} - 1 = 65535$ numbers from -32767 to +32767. So, an n-bit word can represent $2^n - 1$ numbers from $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$.

(iii) 2's Complement Representation

The 2's complement representation is a method used to represent signed integers in binary form. It is widely used in computers because it allows easy arithmetic operations like addition and subtraction using the same hardware for both positive and negative numbers. In this system, a positive number is represented as its normal binary equivalent, while a negative number is obtained by taking the 1's complement of its binary equivalent (inverting all bits) and then adding 1 to the least significant bit. For a given word length, the binary number with MSB 1 followed by 0s will be of no use in the previous two representation methods. But 2's complement representation eliminates this problem and simplifies binary calculations. The steps required to find the 2's complement form of a decimal number are given below.

1. Write the binary equivalent of the absolute value (magnitude) of the given number and express it using the required word length.
2. If the decimal number is positive, write the binary form obtained in step1 as the 2's complement of the given number.
3. If the given number is negative, find the 1's complement of this binary number and add 1 to the least significant bit (rightmost bit) of the 1's complement. (Refer to section 2.3 to perform addition operation).
4. The result obtained is the 2's complement of the given number.

Example: Represent +75 in 2's complement form.

Binary equivalent of the magnitude (75) = 1001011

Binary number of 75 in 8 bits = 01001011

Thus, +75 in 2's complement form = (01001011)₂

0	1	0	0	1	0	1	1
---	---	---	---	---	---	---	---

Example: Represent -75 in 2's complement form.

Binary equivalent of the magnitude (75) = 1001011

Binary number of 75 in 8 bits = 01001011

-75 in 1's complement form = 10110100

Adding 1 to this number +1

1	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---

Thus, -75 in 2's complement form = (10110101)₂

Table 2.4 explains why the 2's complement method is preferred in modern computers as it eliminates the negative zero issue. Let us use a 3-bit word length for easier understanding.



In 2's Complement form, an 8-bit word can represent $2^8 = 256$ numbers from -128 to $+127$. Similarly, a 16-bit word can represent $2^{16} = 65536$ numbers from -32768 to $+32767$. So, an n -bit word can represent 2^n numbers from -2^{n-1} to $+(2^{n-1} - 1)$.

Another important fact to be noted is that the MSB is 0 for positive numbers and 1 for negative numbers in all the three representations.

Table 2.4: Integer representation

Decimal Number	Sign & Magnitude	1's Complement	2's Complement
+0	000	000	000
+1	001	001	001
+2	010	010	010
+3	011	011	011
-0	100	111	-
-1	101	110	111
-2	110	101	110
-3	111	100	101
-4	-	-	100

Binary Subtraction Using Complements

Binary subtraction using complements is a method that simplifies the process of subtracting one binary number from another. Instead of using the traditional subtraction rules (which involve borrowing), this method converts subtraction into addition by using the complement of the subtrahend (the number to be subtracted). This approach is faster and easier for computers to perform, as the same hardware used for addition can also handle subtraction. Thus, using complements for binary subtraction improves efficiency and reduces the complexity of arithmetic operations in digital systems.

(i) Subtraction Using 1's Complement

The following steps are to be carried out to perform binary subtraction using 1's complement.

1. Add 0s to the left of the subtrahend, if necessary, to make the two numbers have the same number of bits.
2. Find the 1's complement of the subtrahend (the number to be subtracted).
3. Add this 1's complement to the minuend (the number from which the other number is subtracted).
4. If there is a carry bit, out of the MSB of the sum, add this carry to the LSB of the sum.

Example: Subtract $(1001)_2$ from $(1101)_2$ using 1's complement method.

1's complement of the subtrahend, $1001 = 0110$

Adding it with the minuend, $1101 = 1101 +$

Carry bit	1	$ \begin{array}{r} 0110 \\ \hline 1101 + \\ \hline 0110 \\ \hline 1 \\ \hline 0100 \end{array} $
-----------	---	--

Adding the carry bit

You can observe that the result is the same as the one we obtained in the example given in section 2.3.

(ii) Subtraction Using 2's Complement

The following steps are to be carried out to perform binary subtraction using 2's complement.

1. Add 0s to the left of the subtrahend, if necessary, to make the two numbers have the same number of bits.
2. Find the 2's complement of the subtrahend.
3. Add this 2's complement value to the minuend.
4. If there is a carry bit out of the MSB of the sum, ignore it.

Example: Subtract $(1001)_2$ from $(1101)_2$ using 2's complement method.

$$\begin{array}{rcl}
 \text{1's complement of the subtrahend, } 1001 & = & 0110 \\
 \text{Adding 1 to get the 2's complement} & & \underline{\quad + 1 \quad} \\
 \text{2's complement of the subtrahend} & = & 0111 \\
 \text{Adding it to the minuend, } 1101 & = & 1101 + \\
 & & \underline{\quad 0111 \quad} \\
 & & 10100 \\
 \text{Ignore the Carry bit} & \swarrow & \\
 \text{The result is} & & \underline{\underline{0100}}
 \end{array}$$



Let us do

1. Represent the following numbers in the three representation methods.
(a) -78 (b) 89 (c) -123
2. Perform the following binary subtractions using complements method.
(a) $(10011)_2 - (1101)_2$
(b) $(100001)_2 - (11111)_2$
(c) $(10101010)_2 - (10101)_2$
(d) $(11000110)_2 - (1011101)_2$
(e) $(10101000)_2 - (101111)_2$

2.4.2 Representation of Floating Point Numbers

In computers, floating-point representation is used to store and process real numbers – that is, numbers that have a fractional part. These numbers can be extremely large or extremely small. Examples include 3.14, 0.00045, 2500000.0. Representing such numbers accurately is important for scientific, engineering and mathematical calculations.

The term *floating point* means that the **binary point** (similar to the decimal point in decimal numbers) is not fixed at one position. Instead, it can "float" to the left or the right, depending on the size of the number. This flexibility allows computers to represent a very wide range of values – from tiny fractions to huge quantities – using a fixed number of bits. The binary number 1011.11 can be written as 1.01111×2^3 . Here the binary point floats towards left.

Floating-point numbers are written in a form similar to scientific notation. For example, the decimal number 3.45×10^4 can be represented as a base number (called the **mantissa**) multiplied by a power of 10 (the **exponent**). In binary, the same idea is used, except that the base is 2 instead of 10. So, a binary floating-point number is represented as:

$$N = (-1)^{\text{sign}} \times (1.\text{mantissa}) \times 2^{\text{exponent}}$$

In computer memory, a floating-point number is divided into three parts:

1. Sign bit – indicates whether the number is positive (0) or negative (1).
2. Exponent – determines the position of the binary point.
3. Mantissa – represents the actual digits of the number in binary form.

The binary number $(101.01)_2$ can be written as 1.0101×2^2 . It can be expressed in the above format as $(101.01)_2 = (-1)^0 \times (1.0101) \times 2^2$

The most commonly used method for storing floating-point numbers is the **IEEE 754 standard**, which defines two formats:

- Single precision (32 bits) – 1 bit for sign, 8 bits for exponent, 23 bits for mantissa.
- Double precision (64 bits) – 1 bit for sign, 11 bits for exponent, 52 bits for mantissa.

This representation allows computers to handle calculations involving real numbers with good precision and a very large range. However, since only a limited number of bits are available, some decimal fractions cannot be represented exactly in binary, which may lead to tiny rounding errors in calculations. Despite this, floating-point representation provides a practical and efficient way for computers to work with real-world numerical data.

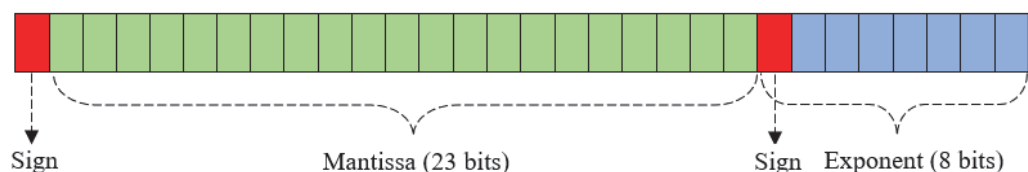


Figure 2.2: Representation of floating point number in 32-bit format



The IEEE (Institute of Electrical and Electronics Engineers) is the world's largest professional organisation dedicated to advancing technology. It is a global association of engineers, scientists, and researchers working in areas such as electronics, computer science, electrical engineering, and communication technologies. The IEEE develops international standards, publishes research journals, organises conferences, and promotes innovation in emerging fields.

The IEEE 754 standard is a rule that tells computers how to store and handle real (fractional) numbers. It makes sure that all computers represent numbers in the same way.

In this system, every number is stored in three parts —

- a sign bit to show positive or negative,
- an exponent to show how large or small the number is, and
- a mantissa (or fraction) to show the actual digits of the number.

IEEE 754 is used for both single precision (32-bit) and double precision (64-bit) numbers. It helps computers perform calculations on very large and very small numbers accurately and consistently.

2.4.3 Representation of Characters

In a computer system, not only numbers but also letters, symbols and special characters need to be stored and processed. Since a computer can understand only binary digits (0 and 1), every character – such as letters (A–Z, a–z), digits (0–9), punctuation marks, and control symbols – is represented by a unique combination of bits. To make this uniform across all devices, standard coding systems are used. These codes assign a specific binary value to each character, allowing computers to store, display, and exchange text data accurately between different systems and applications. Let us have a close look at these coding systems.

(a) ASCII

ASCII, pronounced as 'As-key', stands for American Standard Code for Information Interchange. It is one of the most widely used systems for representing characters in computers. It was developed in the early 1960s to provide a standard way for computers and communication devices to represent and exchange text data. A unique integer number is assigned to each character such as letters, digits, punctuation marks, and control symbols. This number called ASCII code is converted into binary for storing in memory.

Initially, in ASCII, every character was represented by a 7-bit binary code, which means it could represent 128 different characters (from 0 to 127 in decimal). These include:

- Control characters (0–31): Used for non-printing purposes like carriage return, line feed, tab, or backspace.
- Printable characters (32–126): Include letters (A–Z, a–z), digits (0–9), punctuation marks, and special symbols.
- DEL (127): Used as a delete or erase character.

Later, an extended ASCII version was developed using 8 bits (1 byte) instead of 7 bits, allowing up to 256 characters. This extension included additional graphic symbols, accented letters, and special characters used in different languages. ASCII became the foundation for text representation in almost all computer systems and programming languages. Although newer standards like **Unicode** are now used to support a much larger range of global characters and symbols, ASCII still remains the basic and most important character encoding system, forming the core of modern data representation in computers.

(b) Unicode

Unicode is a modern and universal standard used to represent characters from all languages of the world in computers. It was developed to overcome the limitations of the earlier coding systems like ASCII, EBCDIC (Extended Binary Coded Decimal Interchange Code) and ISCII (Indian Standard Code for Information Interchange), which could represent only a limited set of characters. As computing and communication became global, there was a need for a single coding system that could handle characters from different scripts, symbols and languages; and this led to the creation of Unicode.

The Unicode standard assigns a unique number, called a code point, to every character, regardless of the platform, program, or language. This ensures that a text written in any language can be stored, transmitted, and displayed consistently across different computer systems and devices.

Unicode originally used 16 bits which can represent up to 65536 characters. Nowadays Unicode uses more than 16 bits and can represent over 1,00,000 characters from the world's major languages, mathematical symbols, punctuation marks, emojis, and special signs. It uses different encoding forms to store these characters efficiently:

- **UTF-8** (encoding is character dependent and its size can vary from 1 – 4 bytes)
- **UTF-16** (uses 16 bits per character)
- **UTF-32** (uses 32 bits per character)

UTF stands for Unicode Transformation Format. Among these, **UTF-8** is the most widely used on the Internet because it is compact and compatible with ASCII. It is used in web pages, emails, URLs, social media, etc. Unicode has now become the **global standard** for text representation and is supported by all modern operating systems, browsers, and programming languages. It allows seamless communication and data exchange in multilingual environments, making it an essential part of today's digital world.



Let us do

Fill in the following table that shows the comparison of character encoding standards.

Feature	ASCII	ISCII	UTF-8
Full form			
Developed by			
Bit size			
Characters represented			

2.4.4 Representation of Image, Audio and Video

In addition to numbers and text, computers can also handle multimedia data such as images, sounds, and videos. These types of data are also represented and stored in binary form using 0s and 1s. An image is stored as a collection of tiny dots called pixels, each having a specific colour value. Sounds are recorded as samples of sound waves, and videos are stored as a sequence of image frames combined with sound. By converting all these forms of information into digital (binary) form, a computer can store, process, and transmit multimedia data efficiently.

(a) Image Representation

An image is a visual representation of objects, scenes, or patterns. In a computer, an image is stored in digital form using binary digits (0s and 1s). As mentioned earlier, every image is made up of a large number of tiny dots called pixels (picture elements). Each pixel represents a very small portion of the image and has a specific colour value stored as a binary number. The clarity or detail of an image depends on:

- **Resolution:** The number of pixels in an image, usually expressed as width $\times$ height (for example, 1920×1080). A higher resolution means more pixels and better image quality.
- **Colour depth:** The number of bits used to represent the colour of each pixel. For example, in 8-bit colour, each pixel can represent 256 different colours; in 24-bit colour, each pixel can represent about 16 million colours.

There are two main types of digital images:

- **Bitmap (Raster) images:** Made up of pixels arranged in rows and columns. Examples: photographs, scanned pictures.
- **Vector images:** Made up of mathematical shapes like lines and curves. They are resolution-independent and can be resized without losing quality. Examples: logos, diagrams.

Different file formats are used to store digital images. Some use compression to reduce file size. Examples include BMP (Bitmap), JPEG (Joint Photographic Experts Group), PNG (Portable Network Graphics), GIF (Graphics Interchange Format), TIFF (Tagged Image File Format), WEBP (developed by Google), HEIF/HEIC (High Efficiency Image File) and SVG (Scalable Vector Graphics).



Nowadays, you are familiar with various image resolutions like HD, 2K and 4K. The following table shows the quality levels of popular screen resolutions.

Resolution Type	Pixels (Approx.)	Quality	Common Usage
HD	1280 x 720	Good	Smart phones, Monitors
Full HD	1920 × 1080	Good	TVs, Computers
2K	2560 × 1440H	Better	Cinema, Gaming
4K	3840 × 2160	Ultra High	Smart TVs, Professional video

(b) Audio Representation

Sound is a form of energy that travels in the form of continuous waves. However, computers can store data only in digital form — that is, using 0s and 1s. To store sound in a computer, it must first be converted from analog to digital form. This process is called digitisation. The analog sound wave is measured at regular intervals of time. Each measurement is called a sample, and the number of samples taken per second is called the sampling rate. The most common sampling rate for music is 44.1 kHz, which means 44,100 samples are taken every second. Each sample is stored as a binary number that represents the sound level at that instant. The number of bits used to store each sample is called the bit depth. A higher sampling rate and bit depth provide better sound quality but require more storage space.

Various file formats are used to store digital sound. Some are uncompressed, while others are compressed to reduce file size. Examples include WAV (Waveform Audio File Format), AIFF (Audio Interchange File Format), MP3 (MPEG Layer 3), WMA (Windows Media Audio), and AAC (Advanced Audio Coding).

(c) Video Representation

A video is a collection of moving images shown one after another, usually along with sound. Each image in a video is called a frame. When these frames are displayed quickly (generally 24 to 30 frames per second) our eyes see it as continuous motion. Each frame of a video is stored as a digital image, and the accompanying sound is stored as digital audio. Like images, every frame is made up of tiny dots called pixels, and each pixel has a binary value representing its colour. The quality of a video depends on the following.

- **Frame rate:** Number of frames displayed per second (fps). Higher frame rate gives smoother motion.
- **Resolution:** Number of pixels in each frame (e.g., 1920×1080). Higher resolution means better clarity.
- **Bit rate:** Amount of data used to store one second of video. Higher bit rate means better quality but larger file size.

Since videos require a large amount of storage space, they are usually compressed using special algorithms called codecs (short for coder-decoder). Compression helps reduce file size without losing too much quality. There are many formats used to store and share digital videos. Some common ones are AVI (Audio Video Interleave), MP4 (MPEG-4 Part 14), MOV, WMV (Windows Media Video) and MKV (Matroska Video).

2.5 Boolean Algebra

After studying number systems, data representation, and binary operations, we now move on to Boolean algebra, which explains how computers perform logical reasoning and decision-making. Boolean algebra is a branch of mathematics that works with only two values, 1 and 0, representing true and false respectively. Using logical operations such as AND, OR, and NOT, it helps computers evaluate conditions and choose appropriate actions. These operations are not only essential for writing conditional statements in programming but also play a keyrole in designing, analysing and simplifying digital circuits.

Boolean algebra is widely used in the design of digital circuits, switching systems, and the control logic of computer hardware. It is also important in programming, where logical expressions control loops, conditions, and comparisons. We will discuss more about these concepts in later chapters. By learning Boolean algebra, we can understand how complex operations inside a computer are built from simple logical rules, making it an essential concept in Computer Science.

2.5.1 Boolean Operators and Logic Gates

Boolean algebra uses three fundamental logical operations **AND**, **OR** and **NOT** to combine and manipulate binary values (0 and 1). Each operation follows specific rules that determine the result based on the given inputs. These rules are depicted in a table called Truth Table.

A **Truth Table** is a tabular method used to show all possible input combinations and their corresponding output for a logical operation or circuit. It helps in understanding how a Boolean expression or logic gate behaves for every possible input condition. Each row in a truth table represents a unique combination of input values (0s and

1s), and the output column shows the result of applying the logical operation. The inputs are identified as operand variables and the result will be a Boolean expression, which consists of operands and operator. A Boolean expression with m operands and n operators requires 2^m rows and $m+n$ columns.

A **logic gate** is an electronic circuit that performs a specific logical operation based on one or more input signals. Each gate has a specific symbol with the required inputs and an output that follows the rules of Boolean algebra. Logic gates are the building blocks of digital systems such as microprocessors, memory units, and control circuits.

(a) OR Operation and OR Gate

The OR operator performs logical addition and the symbol used for this operation is + (plus). The expression $A + B$ is read as "A OR B". Here, A and B are assumed as the inputs (operands) and $A+B$ is the output (result). Table 2.5 is the truth table that contains the rules of

Table 2.5: OR operation

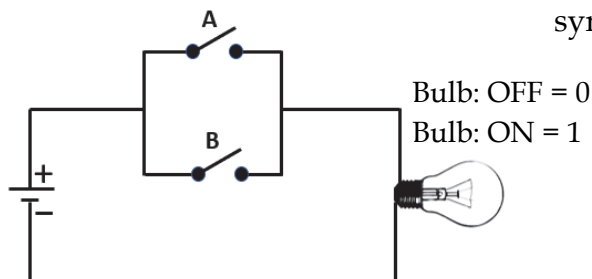
A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

OR operation. The rule of OR operation can be summarised as: if any one of the inputs is 1 (True), output will be 1 (True).

While designing logic circuits, the gate used to implement logical OR operation is known as OR gate. Figure 2.3 shows the OR gate symbol.



Figure 2.3 OR gate



Switch A: Open = 0 (OFF), Closed = 1(ON)

Switch B: Open = 0 (OFF), Closed = 1(ON)

Figure 2.4: Circuit with two switches and a bulb connected in parallel (OR operation)

The working of this gate can be illustrated with an electrical circuit as shown in Figure 2.4. Here A and B denotes two switches. If either of them is ON (1), the bulb will glow, i.e., in ON state (1). This exactly resembles the function of OR gate.

Table 2.6: AND operation

A	B	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

(b) AND Operation and AND Gate

The AND operator performs logical multiplication, and the symbol used for this operation is a . (dot). The expression $A \cdot B$ is read as "A AND B". In this expression, A and B represent the input values (operands), while $A \cdot B$ denotes the output or result. Table 2.6 shows the truth table that lists all possible input combinations and their corresponding outputs for the AND operation. The rule for the AND operation can be stated as: the output will be 1 (True) only when all the inputs are 1 (True).



Fig. 2.5: AND gate

When designing logic circuits, the gate that performs the logical AND operation is called an AND gate. The symbol representing the AND gate is shown in Figure 2.5.

The working of the AND gate can be demonstrated using an electrical circuit, as shown in Figure 2.6. In this circuit, A and B represent two switches. The bulb glows (ON state - 1) only when both switches are ON (1). This behaviour is identical to the function of an AND gate.

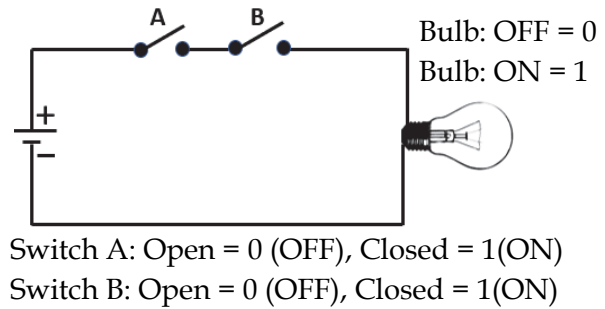


Figure 2.6: Circuit with two switches and a bulb connected in series (AND operation)

OR and AND gates require a minimum of two inputs for their operations.

(c) NOT Operation and NOT Gate

The NOT is a unary operator and hence it requires only one operand. It performs logical negation and the symbol used for this operation is $\bar{}$ (over-bar). The expression $\bar{A}$ is "A bar", where A is the input variable. Sometimes, it is expressed as A' and is read as "A dash". Table 2.7 is the truth table that represents the NOT operation. The rule can be summarised as: the output of the NOT operation will be the opposite value of the input.

Table 2.7: NOT operation

A	$\bar{A}$
0	1
1	0

The gate that performs logical NOT operation is called NOT gate and is shown in Figure 2.7. It is also known as the inverter, because it changes a logic '1' to logic '0' and a logic '0' to logic '1'. It has only one input and one output.



Figure 2.7: NOT gate

A practical example of the NOT operation is an automatic night lamp. During the day, when light falls on the sensor (input = 1), the circuit switches the lamp off (output = 0). At night, when there is no light (input = 0), the lamp automatically turns on (output = 1).



Let us do

1. A table is given below with the basic Boolean operations. Fill in the columns for each combination.

A	B	$A + B$	$A \cdot B$	$\bar{A}$	$\bar{B}$
0	0				
0	1				
1	0				
1	1				

2. Evaluate the expression: $A \cdot B + \bar{B} \cdot C$ for all combinations using a Truth Table.

2.5.2 Boolean Expressions and Principle of Duality

When Boolean variables and/or constants are combined with Boolean operators, Boolean expressions are formed. $(\bar{A} + 1)$ and $(X.Y)$ are examples of Boolean expressions. Each Boolean expression has a dual form which can be derived by

- changing each OR sign $(+)$ to AND sign $(.)$
- changing each AND sign $(.)$ to OR sign $(+)$
- replacing each 0 by 1 and each 1 by 0.

If a Boolean expression is evaluated to 0, its dual expression will be evaluated to 1 and vice versa. For example, the Boolean expression $A + \bar{A}$ is evaluated to 1 for any value of A. Hence, its dual form $A.\bar{A}$ will be evaluated to 0 for any value of A.



Let us do

Write down the dual form of the expressions given in column X into column Y.

X	Y
$A + B.C$	
$(A + B).(A + C)$	
$\bar{A} + \bar{B}$	
$\bar{A} . \bar{B} + 1$	
$A + B . \bar{C}$	
$\bar{A} . B + \bar{B} . C$	

2.5.3 Boolean Identities

When two Boolean expressions are evaluated to the same result for all input combinations, they are considered equivalent. If E1 and E2 represent two expressions which give the same output for any choice of input values, we can say that they are equivalent, and we write $E1 = E2$. This notation of equivalence is known as Boolean identity.

In Boolean algebra, the identities serve as essential tools for simplifying and manipulating Boolean expressions. They help in reducing complex Boolean expressions into simpler forms and thereby simplifying logical circuits, making circuit design easier and more efficient. Let us discuss some commonly used identities.

The Boolean expressions $A . (B + C)$ and $A.B + A.C$ give the same output as shown in Table 2.7. You can verify it from the columns 5 and 8 of this table.

Table 2.7: Expression equivalency checking

A	B	C	$B + C$	$A \cdot (B + C)$	$A \cdot B$	$A \cdot C$	$A \cdot B + A \cdot C$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

So, we can say that the Boolean expressions $A \cdot (B + C)$ and $A \cdot B + A \cdot C$ are equivalent and hence it is a Boolean identity, and we write $A \cdot (B + C) = A \cdot B + A \cdot C$, which is known as the **Distributive law**.

The Boolean expressions $A + 0$ and $A \cdot 1$ can be reduced to A as shown Table 2.8(a) and 2.8(b) respectively. Similarly, $A \cdot 0$ always gives 0 and $A + 1$ always gives 1 as shown in Table 2.9(a) and 2.9(b) respectively.

Table 2.8 (a)

A	0	$A + 0$
0	0	0
1	0	1

Table 2.8 (b)

A	1	$A \cdot 1$
0	1	0
1	1	1

Table 2.9 (a)

A	0	$A \cdot 0$
0	0	0
1	0	0

Table 2.9 (b)

A	1	$A + 1$
0	1	1
1	1	1

Hence the following identities are named **Laws of Identity**.

$$(i) \quad A + 0 = A \qquad (ii) \quad A \cdot 1 = A$$

$$(iii) \quad A \cdot 0 = 0 \qquad (iv) \quad A + 1 = 1$$

Expressions on the LHS of the statements (i) and (ii) are dual to each other. Similarly, the LHS of the statements (iii) and (iv) are dual to each other.

As discussed above, some commonly used identities are known as Boolean laws. These identities are given in Table 2.10.

Table 2.10: Boolean identities

Identity / Law	Statement 1	Statement 2
Idempotent	$A + A = A$	$A \cdot A = A$
Commutative	$A + B = B + A$	$A \cdot B = B \cdot A$
Associative	$A + (B + C) = (A + B) + C$	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$
Distributive	$A \cdot (B + C) = A \cdot B + A \cdot C$	$A + (B \cdot C) = (A + B) \cdot (A + C)$
Absorption	$A + (A \cdot B) = A$	$A \cdot (A + B) = A$
Involution	$\overline{\overline{A}} = A$	

2.5.4 De Morgan's Theorems

Augustus De Morgan (1806–1871), a well-known logician and mathematician from University College, London, proposed two important theorems to simplify complex Boolean expressions. These are popularly known as De Morgan's Theorems. The two theorems are:

- i. $\overline{A + B} = \overline{A} \cdot \overline{B}$
- ii. $\overline{A \cdot B} = \overline{A} + \overline{B}$

In simple terms, these theorems can be stated as:

- i. The complement of the sum of Boolean variables is equal to the product of their individual complements.
- ii. The complement of the product of Boolean variables is equal to the sum of their individual complements.

Table 2.11 proves the first theorem and Table 2.12 proves the second theorem.

Table 2.11: De Morgan's first theorem

A	B	$\overline{A}$	$\overline{B}$	A+B	$\overline{A+B}$	$\overline{A} \cdot \overline{B}$
0	0	1	1	0	1	1
0	1	1	0	1	0	0
1	0	0	1	1	0	0
1	1	0	0	1	0	0

Columns 6 and 7 of table 2.11 proves that $\overline{A+B} = \overline{A} \cdot \overline{B}$

Table 2.12: De Morgan's second theorem

A	B	$\overline{A}$	$\overline{B}$	A.B	$\overline{A \cdot B}$	$\overline{A} + \overline{B}$
0	0	1	1	0	1	1
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	0

Columns 6 and 7 of Table 2.12 proves that $\overline{A \cdot B} = \overline{A} + \overline{B}$



Augustus De Morgan was the first professor of mathematics at University College, London. He introduced the term “mathematical induction” and formulated the famous De Morgan's Laws that are still fundamental in logic and Computer Science today. Interestingly, he was also the first president of the London Mathematical Society, which he helped to establish in 1865.



2.6 Circuit Design

The main objective of Boolean algebra is to simplify Boolean expressions so that they can be easily implemented using logic gates in an electronic circuit. Circuit design is the process of converting a Boolean expression into a combination of logic gates such as AND, OR, and NOT. These gates are connected in such a way that the circuit performs the required logical function. Before going to the simplification process, let us design logic circuits for the given Boolean expressions. By applying Boolean identities, we can minimize the number of gates and connections, making the circuit more efficient, cost-effective, and easier to construct.

2.6.1 Circuit Designing using AND, OR and NOT gates

The three main logic gates AND, OR and NOT are used to build all types of digital circuits. By combining these gates, we can design circuits that satisfy given logical conditions.

Example: Construct the logic circuit for the Boolean expression: $X \cdot Y + \bar{Y}$

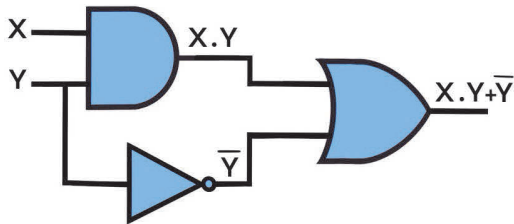


Figure 2.8: $X \cdot Y + \bar{Y}$

Example: Construct the logic circuit for the Boolean expression: $(a + b) \cdot (\bar{a} + \bar{b})$

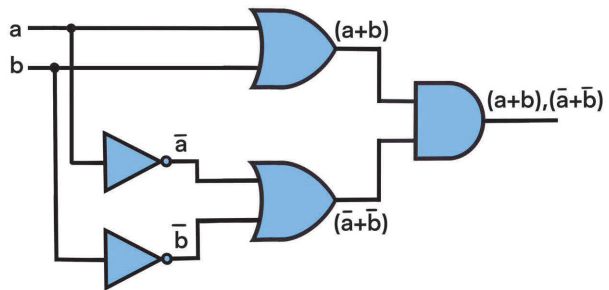


Figure 2.9: $(a + b) \cdot (\bar{a} + \bar{b})$

Consider a classroom lighting system controlled by three switches:

- S1 – the main power switch
- S2 – a safety override switch (must be ON for the bulb to receive power)
- S3 – a manual override switch that can turn the bulb ON even when the safety switch is OFF (used for maintenance)

The bulb will glow only when the main power S1 is ON and either safety override S2 or the manual override S3 is ON. Hence, the Boolean expression for the bulb output B can be written as $S1 \cdot (S2 + S3)$

This means that the bulb will light up (i.e., $B=1$) when the main power switch S1 is ON and at least one of the switches S2 or S3 is ON. The logic circuit will be as shown in Figure 2.10.

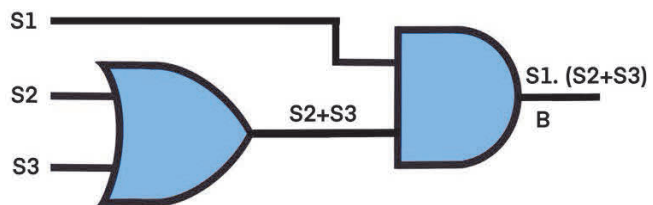


Figure 2.10: $B = S1 \cdot (S2 + S3)$



Let us do

1. Draw logic circuits for the following Boolean expressions:
 (a) $A \cdot B + (A \cdot \bar{C})$ (b) $(A \cdot B) + (\bar{B} \cdot \bar{C}) + (A \cdot \bar{C})$ (c) $(\bar{A} + \bar{B}) \cdot (A + C)$

2. Draw logic circuits for Boolean laws.

3. Consider a Garden Sprinkler Control System as detailed below.

A garden sprinkler works automatically based on three conditions:

- S1 – Power supply switch (must be ON for operation)
- S2 – Soil moisture sensor (ON when soil is dry)
- S3 – Rain sensor (ON when it is raining)

The sprinkler should turn ON only when the power supply (S1) is ON, the soil is dry (S2 = 1), and it is not raining (S3 = 0).

Construct the Boolean expression for this system and draw the logic circuit.

4. Consider a Security Alarm System as detailed below.

A simple alarm system works with the following setup:

- S1 – Main power switch
- S2 – Door sensor (ON when the door is open)
- S3 – Window sensor (ON when the window is open)

The alarm should sound when the power is ON and either the door or the window is open.

Construct the Boolean expression for this system and draw the logic circuit.

2.6.2 Universal Gates – NAND and NOR

In digital electronics, logic gates are the basic building blocks of digital circuits. Among all the logic gates, NAND and NOR gates are called Universal Gates because any other logic gate (AND, OR, NOT, etc.) can be constructed using only NAND or only NOR gates.

This makes circuit design simpler and more flexible. Using a single type of gate reduces the number of different components required, which helps in easy manufacturing, lower cost, and efficient use of space in integrated circuits. It also improves reliability and ease of maintenance, since testing and fault detection become simpler when the same kind of gate is used throughout the circuit. Thus, universal gates provide versatility, simplicity, and cost-effectiveness in the design and construction of digital systems.

Table 2.13: NAND operation

A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

(a) NAND Gate

The NAND gate is a combination of an AND gate followed by a NOT gate. It performs the AND operation first and then inverts (complements) the result. In Boolean algebra, the NAND operation is represented as $\overline{A \cdot B}$, where A and B are the inputs. Table 2.13 is the truth table that contains the rules of NAND

operation. It states that the output is true (1) when at least one of the inputs is false (0) and the output is false (0) only when all the inputs are true (1).

The symbol of the NAND gate is similar to that of an AND gate, but it has a small bubble ($\circ$) at the output end indicating negation (NOT operation) and is shown in Figure 2.11.



Figure 2.11: NAND gate

(b) NOR Gate

The NOR gate is a combination of an OR gate followed by a NOT gate. It performs the OR operation first and then inverts (complements) the result. In Boolean algebra, the NOR operation is represented as $\overline{A+B}$, where A and B are the inputs. Table 2.14 is the truth table that contains the rules of NOR operation. It says that output is true (1) only all the inputs are false (0), and if at least one of the inputs is true (1), the output becomes false (0).

Table 2.14: NOR operation

A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

The symbol of the NOR gate is similar to that of an OR gate, but it has a small bubble ($\circ$) at the output end indicating negation (NOT operation) and is shown in Figure 2.12.

It is already mentioned that any logical operation can be performed using only these gates. Now let us see how the basic logic gates such as AND, OR and NOT can be implemented using only NAND gates or only NOR gates.



Figure 2.12: NOR gate

A minimum of two inputs are required for NAND and NOR gates for their operations.

2.6.3 Implementation of Basic Gates using NAND

Let us see the implementation of the basic gates using NAND gate alone.

(a) NOT Gate Using NAND

We can implement a NOT gate using a NAND by applying the same signal to both the inputs of NAND gate as shown in Figure 2.13.



Figure 2.13: Implementation of NOT using NAND

The truth table shown in Table 2.15 proves this implementation. Columns 3 and 4 show that the NAND and NOT are the same.

Table 2.15: NOT using NAND

A	A.A	$\overline{A.A}$	$\overline{A}$
0	0	1	1
1	1	0	0

(b) AND Gate Using NAND

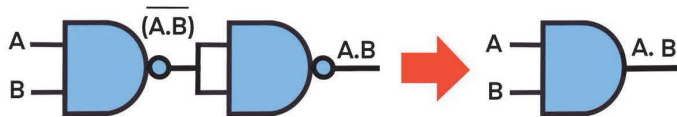


Figure 2.14: Implementation of AND using NAND

We can implement an AND gate using a NAND by connecting a NAND gate followed by another NAND gate to invert the first output as shown in Figure 2.14.

The truth table shown in Table 2.16 proves this implementation. Columns 3 and 6 show that the AND operation and the expression $\overline{(\overline{A.B})} \cdot \overline{(\overline{A.B})}$ constituted by NAND alone are the same.

Table 2.16: AND using NAND

A	B	$A \cdot B$	$\overline{A \cdot B}$	$\overline{(\overline{A \cdot B})} \cdot \overline{(\overline{A \cdot B})}$	$\overline{(\overline{A \cdot B})} \cdot \overline{(\overline{A \cdot B})}$
0	0	0	1	1	0
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	0	0	1

(c) OR Gate Using NAND

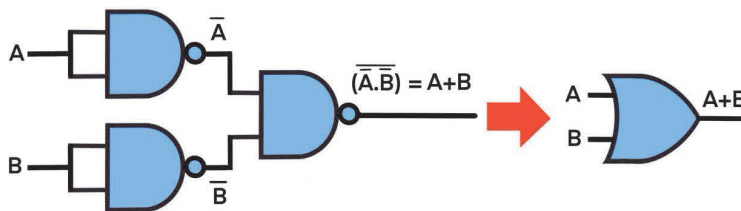


Figure 2.15: Implementation of OR using NAND

The OR gate is replaced by a NAND gate with all its inputs complemented by NAND gate inverters as shown in Figure 2.15.

The truth table shown in Table 2.17 proves this implementation. Columns 5 and 7 show that the OR operation and the expression $\overline{A} \cdot \overline{B}$ constituted by only NAND gates are the same.

Table 2.17: OR using NAND

A	B	$\overline{A}$	$\overline{B}$	$A + B$	$\overline{A} \cdot \overline{B}$	$\overline{A} \cdot \overline{B}$
0	0	1	1	0	1	0
0	1	1	0	1	0	1
1	0	0	1	1	0	1
1	1	0	0	1	0	1

2.6.4 Implementation of Basic Gates using NOR

Now let us see how the basic gates can be implemented using NOR gate alone.

(a) NOT Gate Using NOR

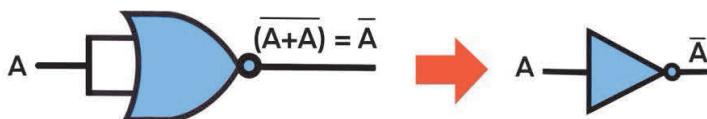


Figure 2.16: Implementation of NOT using NOR

A NOT gate can be implemented using a NOR gate by applying the same signal to both the inputs of NOR as shown in Figure 2.16.

The truth table shown in Table 2.18 proves this implementation. Columns 3 and 4 show that the NOR and NOT are the same.

Table 2.18: NOT using NAND

A	$A+A$	$\overline{A+A}$	$\overline{A}$
0	0	1	1
1	1	0	0

(b) OR Gate Using NOR

We can implement an OR gate using a NOR gate by connecting a NOR gate followed by another NOR gate to invert the first output as shown in Figure 2.17.

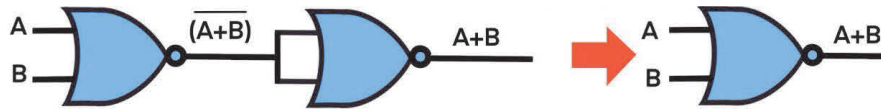


Figure 2.17: Implementation of OR using NOR

The truth table shown in Table 2.19 proves this implementation. Columns 3 and 6 show that the OR operation and the expression $(\overline{A \cdot B}) + (\overline{A \cdot B})$ constituted by NOR alone are the same.

Table 2.19: OR using NOR

A	B	$A + B$	$\overline{A + B}$	$(\overline{A + B}) + (\overline{A + B})$	$\overline{(\overline{A \cdot B})} + \overline{(\overline{A \cdot B})}$
0	0	0	1	1	0
0	1	1	0	0	1
1	0	1	0	0	1
1	1	1	0	0	1

(c) AND Gate Using NOR

The AND gate is replaced by a NOR gate with all its inputs complemented by NOR gate inverters as shown in Figure 2.18.

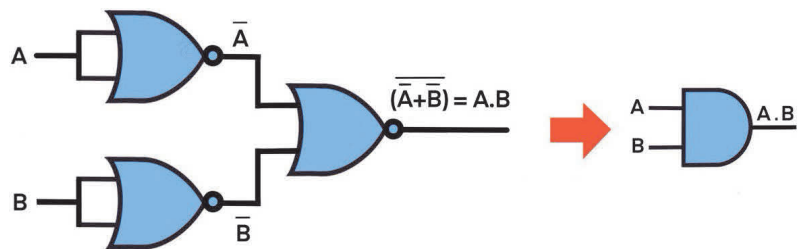


Figure 2.18: Implementation of AND using NOR

The truth table shown in Table 2.20 proves this implementation. Columns 5 and 7 show that the AND operation and the expression $\overline{\overline{A} + \overline{B}}$ constituted by only NOR gates are the same.

Table 2.20: AND using NOR

A	B	$\overline{A}$	$\overline{B}$	$A \cdot B$	$\overline{\overline{A} + \overline{B}}$	$\overline{\overline{A} + \overline{B}}$
0	0	1	1	0	1	0
0	1	1	0	0	1	0
1	0	0	1	0	1	0
1	1	0	0	1	0	1



Let us do

1. Design a circuit using NAND alone and NOR alone for the Boolean expression: $A.(B+C)$. Verify by creating and comparing truth tables.
2. Observe the Boolean expressions given in the following table. For each expression in column A there is an equivalent expression in column B. Create truth table for each expression and find the Boolean identities. Reconstruct the table so that each row contains a Boolean identity.

A	B
$X + Y . Z$	$\bar{X} . \bar{Y}$
$(\bar{X} + \bar{Y})$	$(X+Y).(X+Z)$
$\bar{\bar{X}}$	$Y + X$
$X + Y$	X
$X.(Y + Z)$	$X+(Y+Z)$
$(X + Y) + Z$	$X.Y + X.Z$
$X + (Y.Z)$	$X.(Y.Z)$
$(X.Y).Z$	1
$X + \bar{X}$	$\bar{X} + \bar{Y}$
$\bar{X} . \bar{Y}$	0
$X . \bar{X}$	
$X + X$	

This chapter has explained how data is represented inside a computer and how logical operations are used to process that data. By studying number systems, internal data representation, and Boolean algebra, you gain a clear understanding of how information is stored in binary form and how computers take decisions using logical rules. These concepts help in understanding how both hardware components and software programs work correctly and efficiently. Boolean algebra plays a keyrole in analysing logic gates, which are the elementary building blocks of all digital systems such as processors, memory units and control devices. In the next chapter, *Computer System Organisation*, we will learn about the different parts of a computer system, their functions, and how they work together. This understanding will help us see how computers use these components to perform tasks and solve problems through algorithmic problem solving.

Computer System Organisation

Computer systems are organised into different components that work together to perform tasks efficiently. This chapter explores the major parts of a computer system including the processor, motherboard, memory, input and output devices, and expansion options. It also covers essential software elements such as operating systems, language translators, linkers, loaders, utilities, and applications. This chapter, explains the overall structure and working of a computer system and how hardware and software work together to make it function. Finally, the concept of algorithmic problem solving which explains how computers are used to design step-by-step solutions for computational problems is introduced.

3.1 Components of Computer System

The components of computer system can be broadly categorised into two – **hardware** and **software**.

The electro-mechanical parts of a computer are known as **hardware**. A computer system consists of several physical parts that work together to perform different operations. These components include electronic circuits, memory drives, keyboards, monitors, printers and other devices. **Software** refers to the collection of programs that are executed on computers and electronic devices. Software is generally classified into two – System software and Application software.

System software are those which interact directly with the hardware and provide an environment where users can work with human friendly interfaces. Examples of system software include operating systems such as Linux; compilers such as the GCC (GNU Compiler Collections); interpreters such as the Python interpreter; built in software to operate printers, monitors and other devices, called device drivers; and so on.

Application software are primarily designed for supporting users to perform various routine tasks. They include accounting software, text processing software, browsers, game playing software etc. which we encounter in daily life. Generally, application software use mechanisms provided by the operating system for their working, and rarely interact with hardware

directly. The programs which you will design in Python will also be classified as application software. The distinction between application software and system software cannot be defined completely.

3.1.1 An Overview of Hardware Components

As mentioned above, hardware refers to the physical parts of a computer which include the processor, memory devices, input-output devices. Each component has a specific role, together they help the computer receive data, process information, store results, and provide output. Let us have an overview of these components. Figure 3.1 illustrates the five main functional units of a computer system.

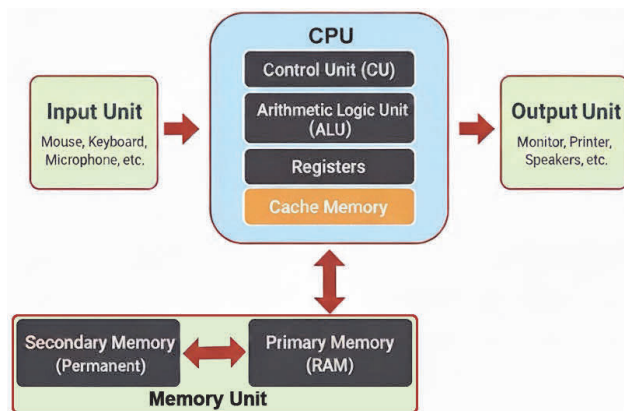


Figure 3.1: Functional units of a computer

Central Processing Unit (CPU):

The central processing unit is the main working engine of a computer system. It contains Control Unit (CU), Arithmetic Logic Unit (ALU) and registers.

Registers: They are tiny, high-speed memory located inside the CPU. They hold data and instructions temporarily while the CPU is processing them. Registers help the CPU work faster by reducing the need to access slower main

memory again and again. They are used to store intermediate results, memory addresses, and instructions currently being executed. Since registers are very expensive to build, their number and storage capacity are limited.

Cache Memory: In modern computer systems, cache memory is a high-speed memory unit, often placed in the CPU chip itself, and is primarily used to improve performance. It acts as a buffer that stores frequently accessed data and instructions, thereby reducing the time the CPU spends waiting for data and instructions from the slower main memory.

Primary Memory (Main Memory): Primary memory is the computer's main memory that works closely with the CPU. It consists of RAM and ROM, which provide the necessary storage for program execution and proper system start-up respectively.

Secondary Memory (Auxiliary Memory): Since programs and data need to be stored for long-term use, they are kept in secondary storage devices. These devices can store information permanently and do not need power to retain data. Earlier devices used mechanical parts, but modern ones mostly use semiconductor technology. Hard disks, Solid State Drives (SSDs), flash drives, memory cards and CDs are common examples of secondary storage devices.

Input and Output devices (I/O devices): Input devices bring data into the computer, while output devices send information out from the computer. Keyboard, mouse, camera, and microphone are input devices. Monitors, printers, and speakers are output devices.

3.1.2 Program Execution Steps

We have already learned that software enables the hardware to perform specific tasks. Now, let us look inside the CPU to understand what takes place there. A program is made up of a series of instructions. Suppose a program is loaded into the main memory for execution. The instructions of the program need to be executed in sequential order. This is achieved by the processes as shown in Figure 3.2.

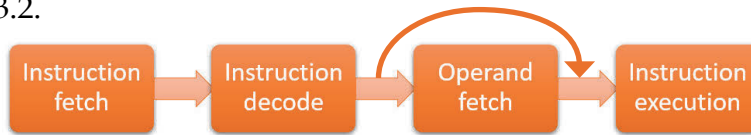


Figure 3.2: Program execution steps

First, the control unit transfers instructions of the program sequentially from the memory to the CPU. This process is called instruction fetch. Once the instruction reaches the CPU, the electronic circuits present in the CPU, called the decoding logic identifies the action to be performed by the instruction. This step is called the instruction decoding step. To execute an instruction, further access to memory might be required. For example, an instruction might require the value stored in a main memory location to be incremented by 1. To execute this instruction, first the value must be transferred from the memory to a CPU. Hence the control unit will initiate a memory read to bring the value into a register in the CPU temporarily. This step is called operand fetch. Once the operand is also fetched, the CPU is ready to perform the action of adding 1 to the operand value fetched. The arithmetic logic unit contains electronic circuits that are capable of doing the addition operation. The control unit directs this circuit to do the required operation. Thus, the execution of the instruction gets completed. The control unit now proceeds to do the next instruction fetch for its execution. If the instruction does not require any data from memory to be processed, the operand fetch step is skipped, and the process moves directly to instruction execution.



Let us do

Answer the following questions.

- Find out which are the system softwares and application softwares that are installed in the computers which you use in your laboratory.
- Which among the following are found inside the CPU of a computer
 - Registers
 - hard disk
 - arithmetic logic unit
 - keyboard
 - RAM

3.2 Hardware Components

This section focuses on understanding the different hardware elements present in a computer system.

3.2.1 Processor

As mentioned earlier, the Central Processing Unit (CPU) is the main component of a computer, and the performance of a CPU determines the overall performance of a computer system. CPUs of modern computers are integrated circuit (IC) packages containing millions of transistors and other components fabricated into a single silicon chip. It is also referred to as a microprocessor. The CPU is considered as the brain of the computer because it performs all operations, controls system activities, processes information quickly, and coordinates tasks. A CPU is usually plugged into a large socket on the main circuit board (the motherboard) of the computer. Since heat is generated when the CPU works, a proper cooling system is provided with a heat sink and fan.



Figure 3.3: Processor

Intel Core i3, Core i5, Core i7, Core i9, Apple M-series, and AMD Ryzen processors are mainly used in desktop and laptop computers whereas Apple A-series, Snapdragon, and MediaTek processors are commonly used in smartphones and tablets, as on 2025.

Although the CPU is the main general-purpose processor of a computer, certain complex tasks can be handled more efficiently by dedicated processors designed for specific functions. This concept is known as heterogeneous computing. Let us examine some of the key examples.

Graphics Processing Unit (GPU): GPUs were originally used to quickly render images and 3D graphics. This processor is designed with hundreds or thousands of simple processors optimised for performing many calculations simultaneously. Their ability to perform thousands of calculations in parallel makes them vital for other tasks as well, especially for training complex AI models.



A neural network is a type of artificial intelligence (AI) model designed to mimic the way the human brain works. It consists of many interconnected processing units called neurons that work together to recognize patterns, make decisions, and learn from data. Neural networks are widely used in tasks such as speech recognition, image processing, self-driving cars, and language translation. They become more accurate over time as they are trained with more data.

Specialised Accelerators (TPU/NPU): For tasks like Artificial Intelligence and Machine Learning, companies have developed highly specialised processors. The Tensor Processing Unit (TPU), for example, is built specifically to handle the complex mathematical operations required by neural networks, providing significantly higher efficiency for AI tasks than a general-purpose CPU or even a standard GPU. The Neural Processing Unit (NPU) is a common accelerator found in modern smartphones and laptops to handle AI tasks quickly and efficiently on the device.

3.2.2 Motherboard

A motherboard is a large Printed Circuit Board (PCB) to which all the major components including the processor are integrated (see Figure 3.4). It also provides expansion slots for adding additional circuit boards like memory, graphics card, sound card, etc. Modern smartphones and tablets use highly integrated motherboards. The desktop motherboards have many separate slots. Motherboards in smartphones have almost all components including the CPU, RAM, and storage integrated onto a single, compact board. This helps to save space and increase efficiency.

The most essential parts on a present-day motherboard include the Central Processing Unit (CPU) which plugs into a dedicated socket, and the memory slots for installing RAM. Communication between the CPU, memory, and peripherals takes place via buses, which are electronic pathways on the board. All operations are synchronised by the system clock, which generates precise timing signals. The BIOS (Basic Input/Output System) chip that starts the computer during power-on and manages the initial hardware configuration, is also integrated on to the motherboard.

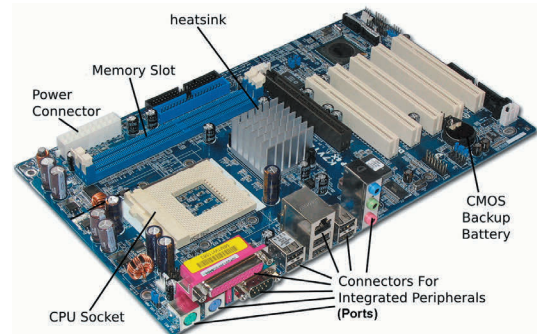


Figure 3.4: Motherboard

3.2.3 Ports and Expansion Slots

Computers have ports and expansion slots that enable connections with external devices and internal upgrades, making the system versatile and expandable.

Ports

Ports are physical connectors located on the computer's exterior or rear panel. They allow peripherals and devices to be connected for data transfer, power supply, or signal transmission. Figure 3.5 shows some ports used in computers. Let us see the purpose of some of them.

USB (Universal Serial Bus) Port:

These are the most widely used ports for connecting a variety of devices such as keyboards, mouse, external drives, printers, and smartphones. The latest versions include USB Type-C, as on year 2025, which supports higher speed. Other USB types are USB Type-A, USB Type-B, Mini-USB, Micro-USB.

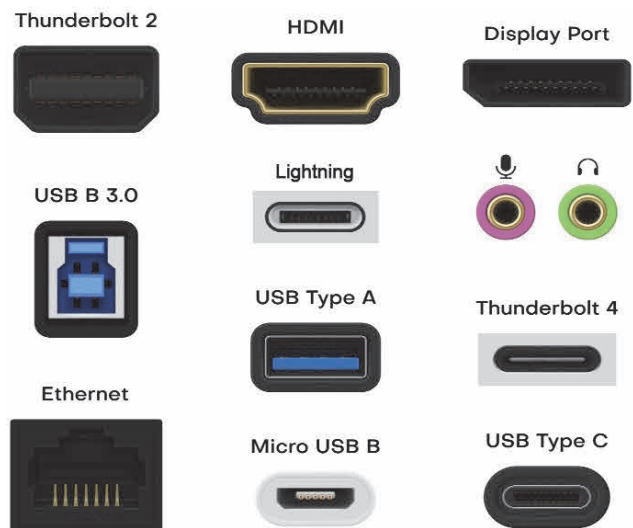


Figure 3.5: Different types of ports

HDMI (High-Definition Multimedia Interface) Port: These are used to connect monitors, televisions, and projectors. It transmits both high-quality video and audio signals through a single cable, making it ideal for multimedia usage. There are different types of HDMI ports for different purposes.

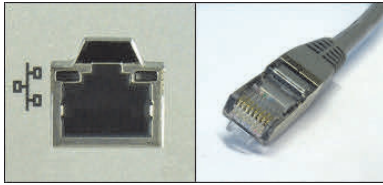


Figure 3.6: LAN port & RJ45

Ethernet Port/LAN port: It is used for connecting computers and other devices to a wired local area network (LAN), providing stable network connectivity. It transmits data packets between network devices using Ethernet cables. The standard connector RJ45 (see Figure 3.6) is used to connect computers to an ethernet port.

Audio Port: It is used to connect audio peripherals like headphones, speakers, microphones, and home theatre systems to a computer.

Thunderbolt Port: This port is a high speed hardware interface developed by Intel in collaboration with Apple for connecting external peripherals to a computer such as high speed external storage drives and external monitors.

VGA (Video Graphics Array) Port: VGA is a standard connection used to transmit analog video signals from a computer or other video source to a display device, such as a monitor or projector.

Display Port: Its primary function is to connect a video source, to a display device like a monitor or projector. Display Port is known for its high bandwidth, which supports high resolutions, high refresh rates, and multiple monitors from a single port.

Expansion Slots

Expansion slots are sockets on the motherboard that allow the installation of additional hardware components called expansion cards, to add or enhance functionalities in desktop or server computer. Some important expansion slots are mentioned below.

PCI Express (PCIe): This stands for Peripheral Component Interconnect Express. It is the current standard, offering high-speed communication between the motherboard and the expansion card.

M.2 Slot: This is a smaller, modern slot, often connected through the PCIe interface, primarily used for installing high speed Solid State Drives (SSDs) or wireless network cards.

AGP (Accelerated Graphics Port): This is an older, legacy slot that was specifically designed to handle dedicated graphics cards before the widespread adoption of PCIe.

By adding an expansion card (like a dedicated graphics card or a high-speed networking card) into an expansion slot, you can customize and improve the computer's capabilities.

3.2.4 Memory

Memory is a place where we can store data, instructions and results temporarily or permanently. Based on the permanent and temporary nature of data storage, memory is classified into volatile and non-volatile. Volatile memory requires a continuous supply of power to retain the stored data. If the power is removed or interrupted, all content stored in this type of memory is lost. Random Access Memory (RAM), cache memory and registers are examples for volatile memory. Non-volatile memory retains its stored data even when the power is turned off.

Read Only Memory (ROM), hard disk, pen drive and memory card are some examples. Before learning more about memory, let us have a look at the different memory measuring units shown in Table 3.1.

Memory can be further divided into primary memory and secondary memory based on how CPU access them.

(i) Primary Memory

Primary memory is the main memory of the computer that the CPU can access directly. It stores data and instructions that are currently being processed. It is faster than secondary memory, but has limited capacity. This includes mainly two types of memory such as RAM and ROM.

RAM (Random Access Memory)

RAM is a type of primary memory that temporarily stores data and instructions while a program is being executed. It allows the CPU to access data and instructions directly. Data or instructions to be processed by the CPU must be placed in the RAM. It is a volatile memory. The speed of a RAM refers to how fast the data in memory is accessed. Its speed is measured in Megahertz (MHz) which indicates how many cycles of read/write operations can be performed on the memory per second. When a computer is in use, its RAM contains the operating system, the application software currently on execution and any data that is being processed. Figure 3.7 shows a RAM chip.

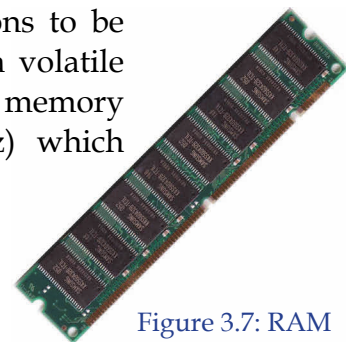


Figure 3.7: RAM

Table 3.1: Computer memory measurements

1 bit	0 or 1 (Binary digit)
1 Nibble	4 bits
1 Byte	8 bits
1 Kilo Byte (KB)	1024 Bytes
1 Mega Byte (MB)	1024 KB
1 Giga Byte (GB)	1024 MB
1 Tera Byte (TB)	1024 GB
1 Peta Byte (PB)	1024 TB
1 Exa Byte (EB)	1024 PB
1 Zetta Byte (ZB)	1024 EB
1 Yotta Byte (YB)	1024 ZB

ROM (Read Only Memory)

When a computer is powered on, it requires some basic instructions to begin its operation, such as checking the hardware and loading the operating system from secondary storage into main memory. For this reason, certain instructions must remain stored even when the



Figure 3.8: ROM

computer is turned off, so that they are available at start-up. To hold these essential instructions, a special non-volatile primary memory is used. This memory is known as ROM and is shown in Figure 3.8.

The CPU can generally perform only read operations on it, and its contents cannot be easily altered. ROM is used in most computers to hold a small, special program known as the Basic Input Output System (BIOS). When the computer is switched on, the BIOS performs a process called Power-On Self-Test (POST) to check whether the main hardware components such as memory, keyboard, and display are working properly. If everything is functioning correctly, the BIOS initiates the process of loading the operating system into RAM. Compared to RAM, ROM is slower.

In many computers, the traditional BIOS is increasingly replaced by Unified Extensible Firmware Interface (UEFI). UEFI performs the same basic functions as BIOS, including hardware initialisation and loading of the operating system, but it offers several advantages. UEFI supports faster booting, better security features such as Secure Boot, larger storage devices, and a more flexible and user-friendly environment. As a result, UEFI improves system startup efficiency, reliability, and security.

ROM comes in different types with varying flexibility. Programmable Read Only Memory (PROM) is manufactured blank and can be programmed only once, while Erasable Programmable Read Only Memory (EPROM) allows erasing with ultraviolet light and rewriting for occasional updates. Electrically Erasable Programmable Read Only Memory (EEPROM), on the other hand, can be electrically erased and reprogrammed multiple times.

Cache Memory

The CPU is much faster than the main memory (RAM). When the CPU requests data from RAM, it often has to wait because RAM cannot supply data at the same high speed. This waiting time reduces the overall performance of the system. To overcome this problem, a small and very fast memory functions as shown in Figure 3.9. This special memory is called cache memory. Cache memory is divided into three levels based on its proximity to the CPU:

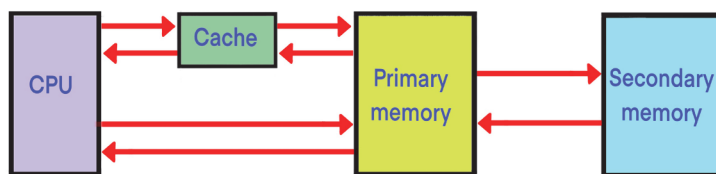


Figure 3.9: Functioning of cache

- Level 1 (L1) Cache: Closest to the CPU, extremely fast but small in size.
- Level 2 (L2) Cache: Larger than L1, slightly slower, but still faster than main memory.
- Level 3 (L3) Cache: Shared among cores in multi-core processors, larger and slower than L2.

Cache memory temporarily stores frequently accessed data, instructions, and intermediate results so that the CPU can access them quickly. When the processor needs to read from or write into a location in RAM, it first checks whether a copy of that data is available in the cache. If the required data is found (called a cache hit), the CPU can read it immediately, which is much faster than accessing it from RAM. Cache memory is faster but more expensive than RAM. However, it is used to maximize system performance by reducing the delay between the CPU and the main memory.



Let us do

Answer the following questions.

1. What is the unit for measuring the speed of a processor?
2. Name any two processors commonly used in desktop or laptop computers and two processors used in smartphones or tablets.
3. Differentiate between volatile memory and non-volatile memory with one example for each.
4. Mention at least three essential components integrated into the motherboard.
5. Explain how cache memory improves the performance of a computer.
6. Is it possible for a computer to execute a program without it being loaded into the RAM? Justify your answer.
7. Explain the role of BIOS in ROM of a computer.
8. Differentiate BIOS and UEFI.
9. Distinguish between RAM and ROM.
10. Arrange the following types of memory in the order of their speed: RAM, Hard Disk, Cache, Registers.

(ii) Secondary Memory

To store large amounts of data and programs permanently, computers use secondary memory, also called auxiliary storage.

Secondary memory is essential in a computer system for several reasons. It provides permanent storage for data and programs, since the contents of RAM are lost when the computer is switched off. It also offers a much larger storage capacity than primary memory at a relatively lower cost, making it suitable for storing a large volume of information. Another important advantage is portability, as devices like pen drives and external hard disks make it easy to carry and transfer large volumes of data between computers.

Secondary memory devices can be classified based on the technology used for storing and accessing data. Different types of secondary storage media use different methods to store information such as magnetic recording, laser technology, or electronic circuits.

Magnetic Storage Devices

These devices use magnetic material to store data. The surface of the storage medium (such as a disk or tape) is coated with a magnetic layer that can be magnetized in different directions to represent binary data (0 and 1). A magnetic head is used to read and write data on the surface. During writing, the head magnetises specific areas to store information, and during reading, it detects these magnetic patterns to retrieve the data. Since they rely on mechanical movement (spinning disks or moving tape), magnetic storage devices are generally slower. However, they offer large storage capacity at a low cost, making them ideal for bulk data storage. Hard Disk Drive (HDD) shown in Figure 3.10 is an example.

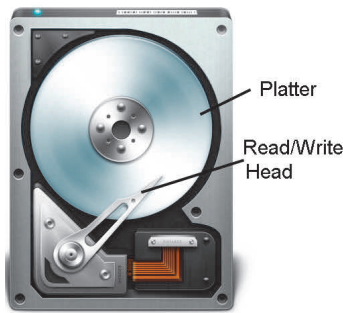


Figure 3.10: Hard disk

Optical Storage Devices

These devices use laser technology to read and write data on the surface of a disk. The surface of the disk is coated with a special reflective material. When data is written, a laser beam creates tiny pits and lands (microscopic marks) on the surface to represent binary data (0s and 1s). During reading, another laser beam scans these marks and converts the reflected light patterns back into electrical signals that the computer can understand. Optical storage devices are mainly used for distributing software, music, videos, and backup data. They are portable and inexpensive, but have limited storage capacity compared to modern magnetic and solid state devices. Examples: Compact Disc (CD), Digital Versatile Disc (DVD), Blu-ray Disc.



Figure 3.11: Optical storage devices

However, their usage has declined significantly due to the growth of faster, more compact, and higher-capacity storage technologies.

Solid State Storage Devices

Solid state storage devices utilise flash memory technology. Flash memory is a type of EEPROM (Electrically Erasable Programmable Read Only Memory), optimised to store large volumes of data permanently. Solid state storage devices have no moving parts because they do not use any rotating disks or read/write heads like magnetic or optical devices. When data is stored or retrieved, electrical signals are used instead of mechanical movement. Solid State Drive (SSD) and USB Flash Drive are examples.



Figure 3.12: Solid state storage devices

Today, flash-based storage and cloud storage services have largely replaced optical media. These newer technologies offer much higher speed, better durability, and easier data transfer options.

NVMe (Non-Volatile Memory Express) Protocol

NVMe (Non-Volatile Memory Express) is a modern communication protocol designed specially for high-speed solid state drives (SSDs) that use PCI Express (PCIe) interface. PCI Express is a very fast connector on the motherboard used to attach high-speed devices like graphics cards and NVMe SSDs. The earlier SSDs used older protocols like Serial Advanced Technology Attachment (SATA), which were originally created for slower hard disks. Because of this, those SSDs could not fully use their actual speed. NVMe overcomes this limitation by allowing the computer and the SSD to communicate directly and much faster through PCIe lanes.

NVMe drives can handle thousands of parallel data requests at the same time, which greatly increases speed and reduces delay. This makes NVMe storage much faster than traditional SATA-based SSDs. As a result, computers using NVMe drives start up quickly, load applications faster, and handle heavy tasks like gaming, video editing, and data processing more efficiently. NVMe has become the preferred standard for high-performance storage in modern computers.



Cloud storage is a modern form of secondary storage in which data is stored on remote servers and accessed through the Internet. Instead of saving data on a local storage device, users can upload, store, and retrieve their files from anywhere using computers or mobile devices. Cloud storage offers high scalability, allowing users to increase or decrease storage space as needed without additional hardware. It provides data backup and recovery facilities, reducing the risk of data loss due to hardware failure. Cloud service providers also implement security measures such as encryption and access control to protect stored data. Examples of cloud storage services include Google Drive, OneDrive, and Dropbox, Apple iCloud, Amazon Drive etc.

3.2.5 Input Devices

A computer receives data and commands through a variety of input devices used for different purposes. Common input devices include the **keyboard** which allows text and commands to be entered, and the **mouse** which helps in pointing and selecting items on the screen. Devices like **touchscreen** and **touchpad** enable direct touch-based interaction, while **joysticks** are often used for gaming and simulations. For capturing sound, a **microphone** is used, and a **scanner** helps convert physical documents into digital form. Specialised devices also play an important role, such as **OMR** for reading marked answer sheets, **MICR** for processing bank cheques, and **digital camera** and **webcam** for capturing images and videos. Modern authentication and security make use of **biometric sensors**, while **smart card readers** help read data stored on chip-enabled cards. Codes and labels are scanned using **barcode readers** and **QR code readers**. Creative tasks often rely on a **stylus pen** and **graphic tablet** for precise writing or drawing. Additionally, advanced input options include **game controllers**, **light guns**, **motion or gesture sensors**, **eye trackers**, **VR controllers**, and **foot pedals**, which are used in specialised fields and modern interactive systems.

			
Keyboard	Mouse	Joystick	Microphone
			
Scanner	Web cam	Biometric sensor	Smartcard reader
			
Bar code reader	Touchpad & Light pen	Game controller	Touchscreen

Figure 3.13: Various input devices

3.2.6 Output Devices

Before processing begins, the computer receives data and instructions from the user through input devices. After processing, the computer produces results that need to be communicated back to the user. Output devices are hardware components that allow the computer to present these results in a form that humans can understand. Let us see some of the commonly used output devices.

Visual Display Unit



Figure 3.14: LED and OLED

A Visual Display Unit (VDU) is an output device that visually presents text and visual information, including both images (graphics) and moving pictures (video). Display devices are of different types based on their technology and usage. These include Liquid Crystal Display (LCD) monitors, Thin Film Transistor (TFT) monitors (a type of LCD with improved image quality), and Light Emitting Diode (LED) monitors, which are energy-efficient flat-panel displays. Advanced devices include Organic Light Emitting Diode (OLED) monitors, used in high-end computers, laptops, and smartphones, offering better contrast and thinner designs. In addition to monitors, projectors are also considered as visual display units, as they project images and videos onto a large screen or surface for presentations, teaching, and entertainment.

Printers

A printer is an output device that produces hardcopy of text, graphics, or images from a computer. The output from a printer is permanent on paper and can be used for records, reports, or presentations. Common types of printers include dot matrix printers

which are mainly used in places like billing counters and offices for printing invoices and multipart forms. Modern printers such as laser printers, inkjet printers, and thermal printers are widely used in homes, offices, shops, hospitals and industries.



Figure 3.15: Printers

Other Output Devices

Output devices such as plotters, 3D printers, and speakers help to present information from a computer in different forms. Plotters are used to produce large, high-precision drawings and graphics, and are mainly available as drum plotters and flatbed plotters, commonly used in fields like engineering, architecture, and map making. 3D printers are advanced output devices that create physical three-dimensional objects from digital models using materials such as plastic, resin, or metal, making them useful in engineering, medicine, and modern manufacturing. Speakers are audio output devices that allow users to hear sound from multimedia applications.



Figure 3.16(a): Plotter



Figure 3.16(b): 3D Printer



Figure 3.16(c): Speaker



Let us do

Answer the following questions.

1. What is the need of secondary memory in a computer system? How does it differ from primary memory?
2. List four common input devices and explain their primary functions.
3. What is the purpose of the NVMe protocol in modern computer memory?
4. What are the different types of visual display units used in modern computers?

3.3 Software

Unlike hardware, which is the physical part of a computer, software is intangible and invisible, which guides the system to work according to our needs. Software can be broadly classified into two – system software and application software.

3.3.1 System Software

System software is a set of programs which aids in the execution of a normal user's computational requirements. The following are some of the major system software components of a computer.

- (a) Operating system
- (b) Language processors
- (c) Linkers and Loaders
- (d) Utility software

(a) Operating System (OS)

OS is a set of programs that controls and supervises the computer hardware and provides services to application software, programmers and users. When a computer is switched on, the operating system is the first program that is loaded onto its memory with the help of BIOS. OS provides a software interface between the user as well as other application software and the computer hardware. DOS, Windows, Unix, Linux, Solaris, macOS, Android, iOS and HarmonyOS are examples.

Functions of an Operating System

Some of the important functions of an operating system can be outlined as follows (see Figure 3.17):

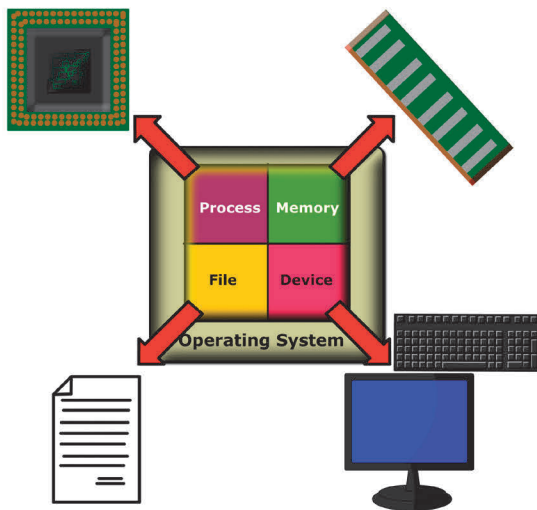


Figure 3.17: OS functions

long is known as **scheduling**. OS manages creation, scheduling and coordination of processes to maximise CPU utilisation.

Memory management: Memory management deals with the efficient use of primary memory (RAM). The operating system decides on how much memory to be allocated to each process and assigns it accordingly. When a process no longer needs memory, the OS deallocates it so that it can be used by other processes. It also ensures that one process does not interfere with memory areas of other processes, maintaining data protection and system stability.

File management: It deals with the way data and programs are stored, organised and accessed on storage devices like hard disks, SSDs or pen drives. The operating system creates and maintains a file system, which helps users store information in the form of files and organise them into folders or directories. The OS is responsible for creating, reading, writing, renaming, copying, moving and deleting files based on instructions from the user. It also manages file permissions and access rights to ensure data security, allowing only authorised users or programs to access specific files.

Device management: Device management is the function of the operating system responsible for controlling and coordinating all the input and output (I/O) devices attached to the computer, such as keyboards, printers, monitors, and storage drives. The operating system communicates with each hardware device through a special software called a device driver. The device driver converts OS commands into device-specific commands. The OS also keeps track of the status of each device, decides which process can use a device at a particular time, and manages the allocation and deallocation of devices to different processes. The OS also provides interfaces to various networking devices used for communication across computers.

(b) Language Processors

We know that a computer understands instructions in machine code, i.e. in the form of binary digits 0s and 1s. It is difficult for us to write program directly in machine code. Hence programs are written mostly in high level languages such as C++, Java and Python. which are designed to make programming easy for human beings. A program written in any high level language or assembly language is called the **source program** or **source code**. Assembly language is a low level language that is one step above machine code, using symbolic instruction codes (mnemonics like ADD or MOV) to make instructions slightly more readable for humans.

The source code cannot be executed directly by the computer. The source code must be converted into machine language to be executed. The program translated into machine code is known as **object program** or **object code**. The special system software that is used to translate the program written in high level language or assembly language into machine code is called a language processor or a translator program. The language processors can be any of the three types – assembler, compiler or interpreter.

Assembler: It is used to translate the program written in assembly language into machine code. Assembler is highly machine dependent because the instruction codes (ADD, MOV, etc.) and the names of the hardware components (registers) vary for every type of CPU. It forces the assembler to be customised for a specific CPU architecture.

Compiler: It is a language processor that translates a program written in high level language into machine language. It scans the entire program in a single run. If there are syntax errors in the program, the compiler provides a list of error messages along with the line numbers at the end of the scanning process. If there is no error, the compiler translates the program and generates an object file. Translation using a compiler is called compilation. The programming languages that are compiled include C, C++, Pascal, Go, etc. The entire program is compiled into a stand-alone object file (machine code). This makes the execution of the program generally faster than interpreted programs.

Interpreter: Interpreter is another kind of language processor that converts a program written in a high level language into machine language line by line. The resultant machine code for the first line is executed by the CPU before the next is translated. If there is an error in any line, the interpreter reports the error to the user and the execution of the program is terminated immediately. Python, JavaScript and BASIC are examples of interpreted languages. Since the code is translated and executed line by line, execution is generally slower than compiled programs.

Hybrid Execution: The Bytecode Model

The Hybrid Execution Model, used by languages like Java, Python and C#, employs a two-step translation process to achieve both high portability and efficient run-time performance. First, the source code is translated by a compiler into an intermediate code called bytecode. This bytecode is then loaded and managed by a specialised software layer known as the Virtual Machine (VM). The VM executes the bytecode, initially through interpretation. Crucially, the VM contains a Just-In-Time (JIT) compiler which dynamically translates frequently executed sections of the bytecode into the host CPU's native machine code while the program is running. This optimised approach allows the same bytecode to run efficiently on any machine that has the appropriate VM software installed.

(c) Linker and Loader

The linker is a system program that runs after the compiler or assembler has translated a program into one or more machine readable object files. Its main role is to take separately compiled object files from different parts of a large program, and merge them with any required pre-compiled code from external libraries (like math or input/output routines), to produce one complete executable program.

The loader is a component of the operating system (OS) that takes the final, complete executable file produced by the linker and prepares it to run by setting up an executable process in the memory of a computer. It allocates the necessary space in the computer's main

memory (RAM), copies the program's machine instructions and data from the executable file on the disk and loads them into the allocated memory. Finally, the loader transfers control of the CPU to the program's starting instruction, initiating its execution.

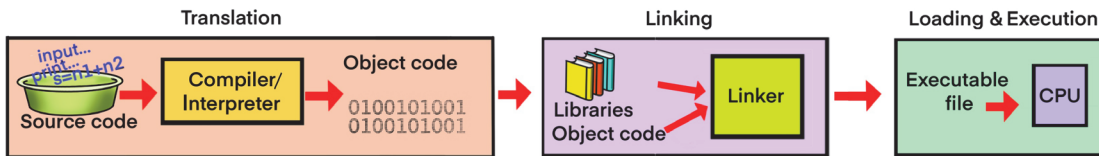


Figure 3.18: Program execution flow/Role of Linker and Loader

Figure 3.18 shows the flow of process during the execution of a program.

(d) Utilities

This term refers to a collection of system software that performs specific tasks that aid the proper maintenance and smooth functioning of a computer system. The following are some important utility software.

File management tools: These tools help users to create, copy, delete, rename, move and organise files and folders efficiently. These utilities also assist in storing, indexing, searching, and sorting files on the system. They make it easier for users to access and manage data in an organised way. Examples are Windows File Explorer, macOS Finder, Google Desktop.

Antivirus software: It is a security program designed to detect, prevent and remove malicious software such as viruses, worms, trojans, spyware and ransomware. It continuously monitors files, applications and system activities to identify suspicious behaviour. Norton, Bitdefender, McAfee, Avast, AVG, Avira, Kaspersky, etc. are some examples.

Backup tools: Backup tools create a duplicate copy of data stored on a disk periodically so that it can be recovered in case of disk failure, data corruption or accidental deletion. Backups ensure that critical files are safe and can be restored whenever needed.

Disk cleaner: A disk cleaner utility scans the storage device for unnecessary files such as temporary files, cache, recycle bin data and other system junk. It also identifies files that have not been accessed or used for a long time and may be occupying a large amount of disk space. The utility then prompts the user to delete such files to free up space and improve system performance.

Compression tools: These are utility software used to compress large files so that they occupy less storage space. The compressed files can be decompressed back to their original form whenever needed. The process of compressing files is also called zipping, and restoring them to the original state is called unzipping. Common

examples of file compression tools include WinZip, WinRAR and 7-Zip.

3.3.2 Application Software

Application software is designed to perform tasks for users and is categorised into two types: general purpose and customised. General purpose software supports common, everyday activities, such as word processing, creating presentations, working with spreadsheets and managing databases. Customised software, on the other hand, is developed to meet specific user needs. Examples include inventory management systems, school management systems, payroll software, financial accounting systems, hotel management tools, reservation systems, specialised applications like weather forecasting software, etc.



Let us do

Answer the following questions.

1. Classify each of the following as system software, application software and utility software:
operating system, word processing software, anti-virus software, disk-defragmentation software, spreadsheet preparation software, compiler.
2. How do compilers and interpreters differ in their way of translating high level language programs to low level programs? Give one advantage of a compiler over an interpreter and one advantage of an interpreter over a compiler.
3. What is the reason for using intermediate byte code during code translation of certain languages? Is there any disadvantage in using intermediate byte code over directly translating high level code to object code of the target machine?
4. Differentiate between the roles of the linker and the loader in the execution process of a program.
5. List and briefly explain the functions of any four important utility software tools.
6. Write down the latest hardware and software configuration or specification to purchase a new computer.

3.4 Phases in Programming

Programming is the process of writing instructions that tell a computer how to perform tasks. Now that we understand the hardware and software components of a computer, programming becomes the key to making them work together effectively to solve different problems. Writing a computer program is a systematic procedure that starts with identifying a problem and ends with obtaining a correct and working solution. This development takes place through a sequence of well-organised steps, known as the phases of programming, as shown in Figure 3.19. Each phase plays a specific role and helps the programmer move closer to the final goal – a fully functional executable program.

Problem Specification

The programmer must fully understand and define the problem that the program is intended to solve. This involves clearly identifying three things: the input (what data is available), the process (what operations are needed to transform the data), and the expected output (what the final result should look like).

Algorithm Design: Deriving steps to obtain a solution

After identifying the problem, we have to derive the steps involved in solving the problem. This phase is focused on designing the solution's logic for solving the problem. Algorithms and flow charts are the tools to describe the steps in a solution and the flow of control between them.

An algorithm is a structured, step-by-step description of how to solve the problem. It is usually written in a natural language such as English. The flowchart is a visual, diagrammatic representation of the algorithm showing the flow of instructions. Algorithm and flowchart act as the blueprint of code.

Coding

In this phase, the logic defined in the algorithm or flowchart is expressed into a specific programming language code (e.g., Python, C++, Java). The process of writing program instructions in high level language to solve a problem is called programming or coding. Text editors are used for writing the code. The program written in a high level language (HLL) is known as source code.

Translation

Once the source code is prepared, it should be translated using the appropriate language processor. Translation is the process of converting a program written in high level language (source code) into its equivalent version in machine language (object code). As mentioned earlier, a compiler or interpreter is used for this purpose. During this step, the syntax errors of the program will be displayed. These errors are to be corrected by opening the file that contains the source code. The source code is again given for compilation (translation).

Debugging

Debugging is the stage where programming errors are detected and corrected. Programming errors are known as 'bugs' and the process

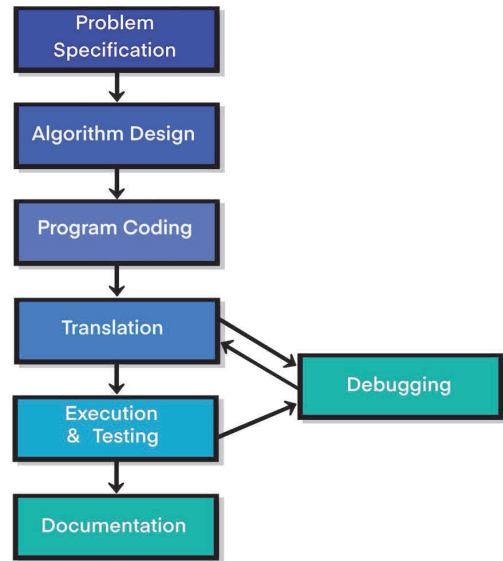


Figure 3.19: Phases of programming



Figure 3.20: Translation process

of detecting and correcting these errors is called debugging. In general, there are three types of errors that can occur in a program – syntax errors, logical errors and run-time errors. **Syntax errors** occur when the rules or syntax of the programming language are not followed. Syntax errors are generally identified by the compiler/interpreter automatically, and hence require only corrections from the programmer.

The second type of error, named **logical error**, is due to the improper planning of the program's logic. The language processor successfully translates the source code into machine code if there are no syntax errors. During the execution of the program, the computer actually follows the program instructions and gives the output as per the instructions. But the output may fail to meet the problem requirements if there are logical errors. Finding logical errors in a program, generally requires careful analysis of the code by the programmer. Meticulous problem specification and algorithm design can prevent logical errors to a large extent.

A **run-time error** is a programming error that occurs during the execution of a program, after it has been successfully compiled or interpreted. This error usually causes the program to terminate unexpectedly. It happens when the program attempts an action that is impossible or illegal for the operating environment to handle at that specific moment, such as division by zero and attempting to open a file that does not exist.

Execution and Testing

Once all errors have been identified and corrected through debugging, the program becomes ready for execution. After running it, the next step is thorough testing, where the program is given different sets of input data that represent a wide range of possible situations. This is done carefully to assess its performance and verify that it consistently delivers accurate and dependable results. This essential stage ensures that the completed program fully satisfies all the requirements established at the beginning of the development process.

Documentation

The documentation phase involves preparing a description of the program. It includes internal documentation such as comments within the code and technical details for developers, as well as external documentation like user manuals and instructions that help users operate the software.

3.5 Algorithmic Problem Solving

After learning the different phases of programming, the next step is to develop a systematic approach to solving problems. Algorithmic

problem solving involves dividing a problem into smaller, logical steps that can be followed to achieve the correct result. An **algorithm** is a step-by-step procedure or set of instructions used to solve a specific problem. To make these steps easier to understand, we can represent them visually using flowcharts. A **flowchart** is a diagram that uses symbols and arrows to show the sequence of actions in a process or algorithm. In this section, we will learn how to design simple algorithms, draw their corresponding flowcharts, and apply both methods to solve sample problems. These skills are fundamental to programming, and in the up-coming chapters, we will begin implementing these solutions using the Python programming language.

When writing an algorithm, we use different kinds of instructions to describe how a problem should be solved step by step. These instructions guide the flow of actions and help convert a real world task into a systematic procedure that a computer can eventually follow. The following are the main types of instructions used in algorithms.

Start and Stop: The Start instruction is the first instruction which denotes the beginning of the algorithm. The Stop instruction is the last one in an algorithm, which marks the termination point.

Input: These instructions specify the data that must be provided before the algorithm begins its processing. The words **Input**, **Read** or **Accept** may be used for this, followed by variables to refer to the data, as in the example: Read A, B.

Process: They describe the operations or calculations to be performed on the input values. Arithmetic operators such as +, -, ×, / are commonly used to indicate the operations. Usually, the result obtained is stored in a variable, as in the example: SUM = A + B.

Assignment: These instructions assign or update the value of a variable, as in the example: TOTAL = TOTAL + N.

Output: These instructions display the result after the processing is completed. The words **Print** or **Write** are used, as in the example: Print SUM.

Conditional or Decision Making: Such instructions are used when a choice must be made based on a condition. They guide the algorithm to take a specific path if a condition is true, and another if it is false. In general, the words **If**, **Else**, **End of If** are used to construct this instruction, as in the example:

```
If (A>B) Then
    Print A, " is greater"
Else
    Print B, " is greater"
End of If
```

If the condition results in True, the instructions given between 'If' and 'Else' will be executed and those within 'Else' and 'End of If' will be skipped. On the other hand, if the condition results in False, the instructions given between 'If' and 'Else' will be skipped and those within 'Else' and 'End of If' will be executed. More about this instruction will be discussed in Chapter 6.

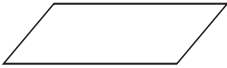
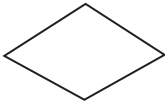
Repetition or Looping: These instructions repeatedly execute a set of steps as long as certain condition is true. They are useful for tasks that require repeated calculations or processing. The words **Repeat** and **While** are used as in the example:

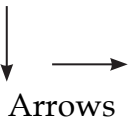
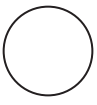
Repeat Steps 2 and 3 While (Condition)
Statement(s)

The Steps mentioned after the word 'Repeat' will be executed repeatedly as long as the 'Condition' results in True. The execution stops when the 'Condition' results in False.

These types of instructions help in writing clear, structured, and logical algorithms. Understanding them will make it easier to design flowcharts and later translate these algorithms into Python programs. Now let us have a brief idea about flowcharts. As mentioned earlier, a flowchart is a pictorial representation of algorithm. Since there is a standard in the structure of an algorithm, a flowchart is drawn using some standard symbols; each represents a type of instruction, as shown in Table 3.2.

Table 3.2: Flowchart symbols and their functions

Symbol	Name	Functional Role
 Ellipse	Terminal	- Represents Start and Stop instructions. - When it is used as Start, an exit-flow will be attached and when it is used as Stop, it follows an entry-flow.
 Parallelogram	Input / Output	- Represents input and output instructions. - It will be attached with an entry-flow and an exit-flow.
 Rectangle	Process / Assignment	- Represents a processing step along with assignment. - It will be attached with an entry-flow and an exit-flow.
 Rhombus	Decision	- Represents a decision based on a condition. - Usually this symbol has one entry-flow and two exit-flows – one branching based on the truth of a condition and the other based on the falsity. - If one of the exit-flows leads backward to an instruction, repetition will take place.

 Arrows	Flowlines	- Indicates the flow of execution of the steps. - The arrow heads indicate the flow, but intersections are to be avoided.
 Circle	Connectors	- Used to connect the flowchart when it spreads over more than one page. - It appears in pair with same label – one with an exit flow and the other with an entry-flow.

Now, let us solve some problems by developing algorithms and constructing flowcharts.

Problem 1: Find the area of a rectangle.

Algorithm:

- Step 1: Start
- Step 2: Input L, B
- Step 3: Area = L x B
- Step 4: Print Area
- Step 5: Stop

Two numbers representing the length and breadth of a rectangle will be given as input to the variables L and B. In step 3, the multiplication is carried out and the result is stored in the variable Area. This result is printed through the instruction in step 4. The flow of execution is visible in the flowchart shown in Figure 3.21.

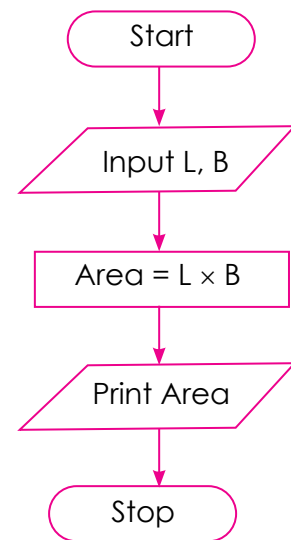


Figure 3.21: Flowchart to find the area of rectangle

Problem 2: Find the greater of two numbers.

Algorithm:

- Step 1: Start
- Step 2: Input N1, N2
- Step 3: If (N1 > N2) Then
- Step 4: Print N1
- Step 5: Else
- Step 6: Print N2
- Step 7: End of If
- Step 8: Stop

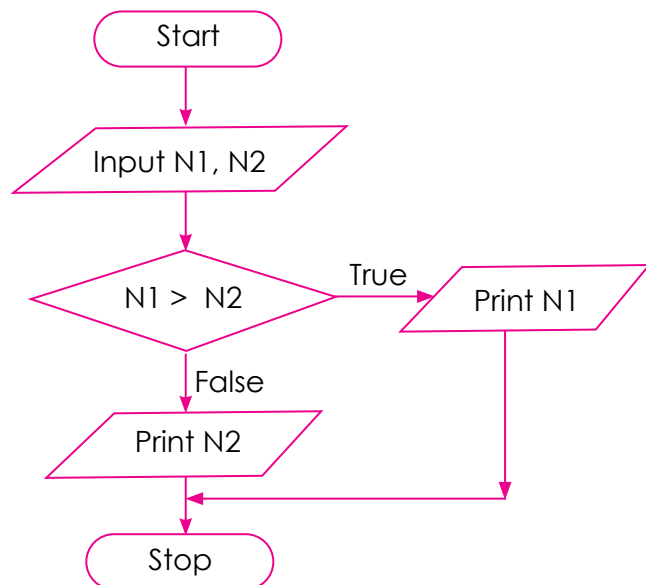


Figure 3.22: Flowchart to find the larger number

In this algorithm, two numbers are given to the variables N1 and N2. In Step 3, the condition (N1>N2) will be evaluated first. If it is found true, Step 4 will be executed and Steps 6 will be skipped. If the condition results in false, Step 4 will be skipped, and Step 6 will be executed. The order of execution can be observed in Figure 3.22.



Let us do

Answer the following questions.

1. Place the following phases of programming in the most appropriate order: Coding, problem specification, debugging, algorithm design, execution and testing.
2. Which is the flow chart symbol for the following purposes?: a) reading a variable from the terminal input. b) an assignment to a variable of the form $a = 3$ c) testing whether the value of a variable a is greater than zero.
3. Write an algorithm and draw a flowchart to find the area of a circle.
4. Write an algorithm and flow chart to read three numbers and find the smallest number among them.

In this chapter, we discussed how a computer is structured and how it performs its tasks, beginning with its major hardware and software components. We learned how a program is executed through the fetch-execute cycle and understood the important role played by various input and output devices. The chapter also presented the phases of programming and the systematic approach required to solve problems using algorithms and flowcharts. By practising a few sample problems, we saw how logical steps can be converted into visual and written forms that a computer can later interpret. With this strong foundation in how computers work and how problems are organised for computation, we are now ready to move on to the next chapter, where we begin writing actual programs using the Python programming language.

Getting Started with Python

We have learnt in the previous chapter that a program is a collection of instructions that a computer executes to perform a specific task. A programming language is used to write these instructions. If these instructions are to be executed by the computer, it should be in machine language form. Since it is difficult for us to write instructions in machine language, high-level languages like C, C++, Java, Python etc. were developed. Though these languages are easier for humans to use, we need translators to convert programs written in high level languages into machine language. We have also learnt that the translator programs are available in the form of compilers and interpreters. Python uses an interpreter to translate its code into machine language, executing it line by line. When a Python program is run, the source code is first converted into bytecode. This bytecode is a low-level set of instructions that the Python interpreter can understand. It is not machine code and cannot run directly on hardware. The bytecode is platform independent, so the same program can run on different systems. It helps Python programs execute faster by avoiding repeated compilation. As a high level language, Python is versatile and widely used in various domains including web development, data science, automation, artificial intelligence and many more.

4.1 Introduction to Python

Python is one of the most popular programming languages, renowned for its simplicity, extensive features, and strong library support. Its simple syntax makes it an ideal choice for beginners, allowing them to focus on programming concepts rather than complex code structures. That is, Python code looks almost like plain English and is easier to read, write and understand. Because of this simplicity, learners can concentrate on the logic of the program instead of struggling with how to write it.



Python was developed by Guido van Rossum, a Dutch computer programmer. He began working on Python in the late 1980s. The first official version, Python 0.9.0, was released in 1991.

He named it after the British comedy group Monty Python's Flying Circus. He was a fan of the show and thought that the name "Python" was short, memorable and a bit quirky, befitting the language's goal of being simple and accessible.



As mentioned above, Python has a vast collection of libraries and frameworks. Libraries are reusable collection of functions, classes, and modules designed for specific tasks such as mathematical data handling, graphics, or networking. We will discuss these concepts more elaborately at a later stage. Frameworks, on the other hand, provide a complete structure for developing software applications. For example, Django and Flask facilitate web development; Pandas, TensorFlow and Scikit-learn support AI, machine learning, and data science. Python supports cross-platform compatibility. That is, Python code can run on multiple operating systems including Windows, Mac and Linux, without requiring significant modifications. Python is open source and hence its source code is freely available for anyone to use, modify, and distribute. This makes Python cost-free, highly flexible, and continuously improved by a large global community. For a given problem, a Python program normally requires fewer lines of code compared to other programming languages like C++ or Java, making Python a preferred choice for many developers.

4.2 Python IDEs

An Integrated Development Environment (IDE) is a software application that provides a comprehensive platform for writing, debugging, and testing code. IDEs enhance productivity, simplify development, and improve code quality, making them essential tools for programmers. You may use any suitable IDE to write your program, but a Python interpreter must be installed on your computer in order to run it.



Figure 4.1: Super key in the keyboard

In this book, we use IDLE version 3.10.12 available with GNU Linux 22.04.5 LTS as the IDE for learning Python. To get the IDLE IDE, follow the steps given below.

- ◆ Press the Super key in the keyboard (refer to Figure 4.1) to launch the search bar in the desktop of GNU Linux.
- ◆ Enter the word 'idle' in the search bar on the desktop as shown in Figure 4.2. The word can be either in upper case or lower case.
- ◆ Click on the IDLE icon, which opens the IDLE Shell as shown in Figure 4.3., which allows us to write instructions that will be executed by the Python interpreter.

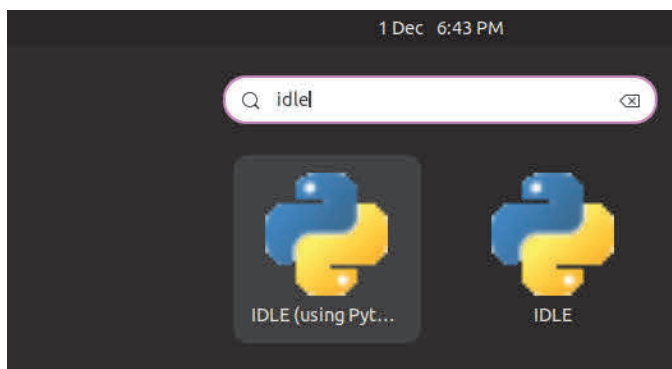


Figure 4.2: Part of Linux desktop with search bar

In the screen shown in Figure 4.3, the symbol `>>>` is the Python prompt, which indicates that the interpreter is ready to execute instructions.

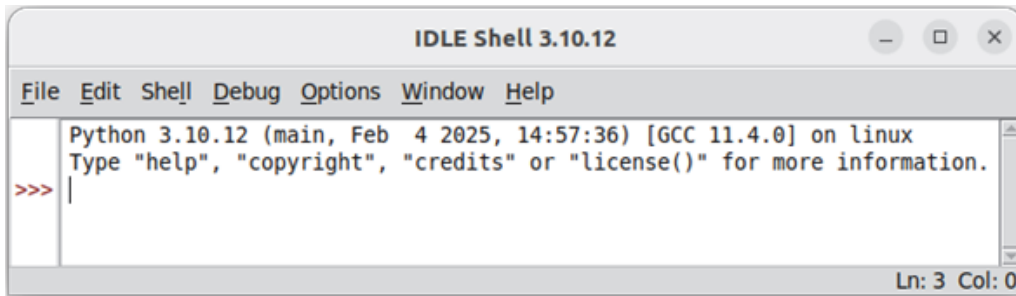


Figure 4.3: Python Shell provided by IDLE

We can type commands or statements on this prompt to execute them using a Python interpreter. There are two modes to use the Python interpreter:

- (a) Interactive mode
- (b) Script mode

The interactive mode allows execution of individual statement instantly, giving immediate results. On the other hand, script mode allows the execution of a file that contains more than one instruction, called script or program.

(a) Interactive Mode

To use the interactive mode, a Python statement is directly typed at the `>>>` prompt and Enter key is pressed. The interpreter will execute the statement and display the result immediately, as shown in Figure 4.4.

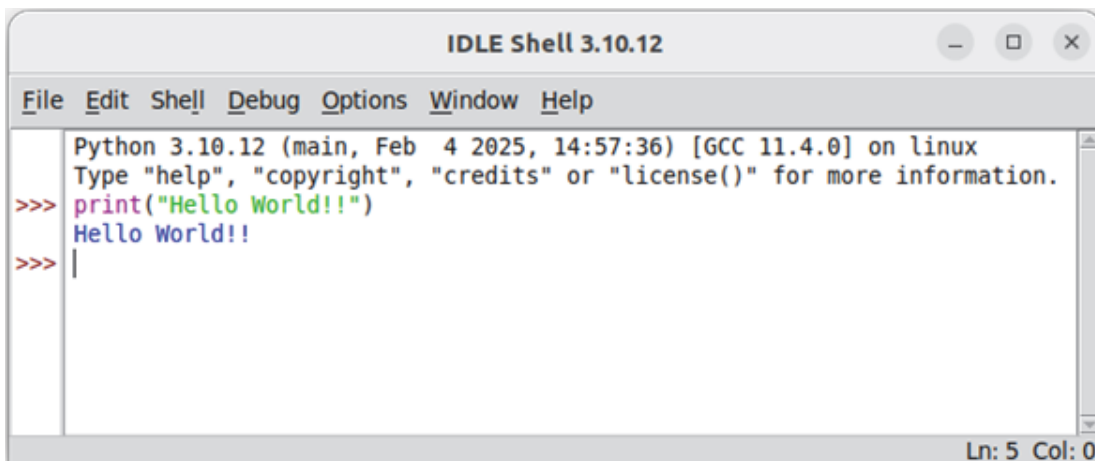
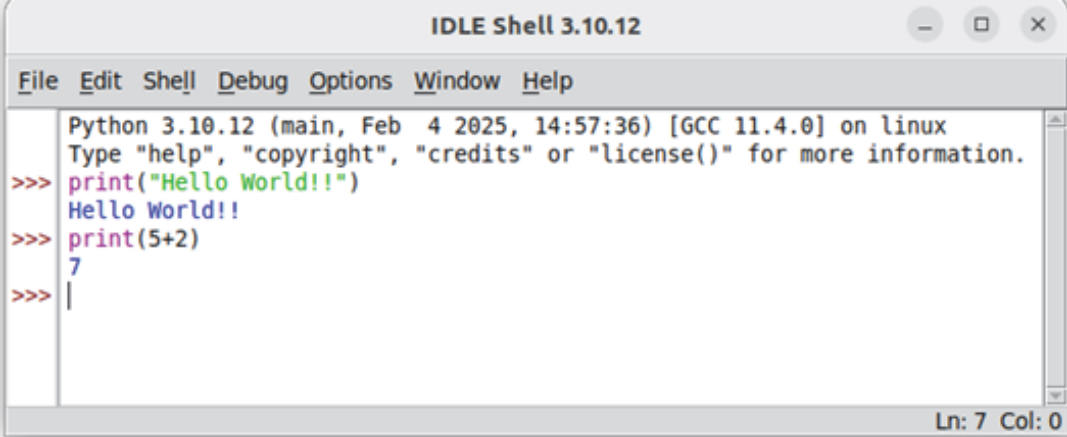


Figure 4.4: Python interpreter in interactive mode

Here, a **print()** function is used as the Python statement to display the message “Hello World!!”. When you press the Enter key after typing the statement, the output (in blue colour) is shown on the next line. After displaying the result, the prompt reappears, allowing you to enter the next command or instruction as shown in Figure 4.5.



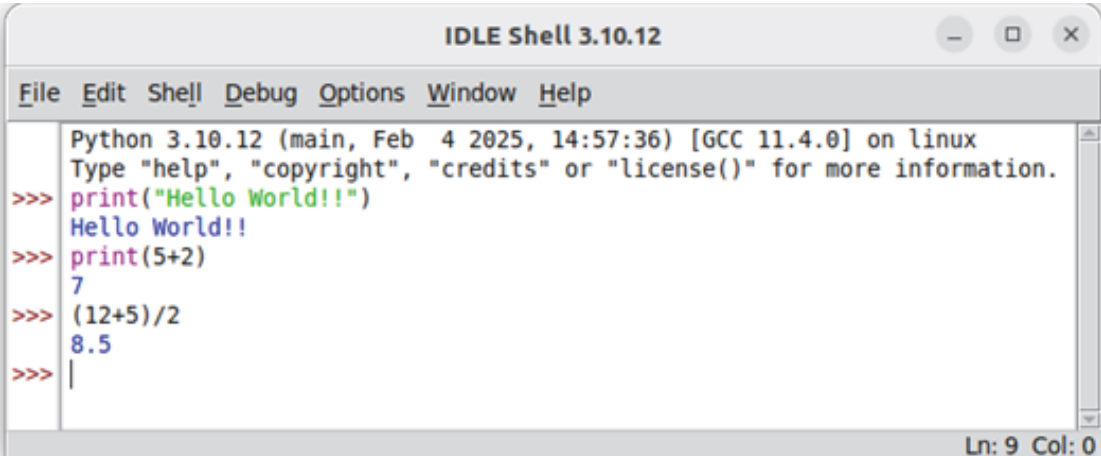
```

Python 3.10.12 (main, Feb  4 2025, 14:57:36) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Hello World!!")
Hello World!!
>>> print(5+2)
7
>>> |

```

Figure 4.5: Interactive use of Python interpreter

In the interactive mode, the IDLE shell acts like a calculator that can perform arithmetic operations such as addition, subtraction, multiplication, and division directly. When you give an instructions to perform an arithmetic operation at the prompt and press Enter key, the result appears on the next line. Figure 4.6 shows the use of IDLE shell as a calculator (refer to the 3rd Python prompt).



```

Python 3.10.12 (main, Feb  4 2025, 14:57:36) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Hello World!!")
Hello World!!
>>> print(5+2)
7
>>> (12+5)/2
8.5
>>> |

```

Figure 4.6: Python interpreter as a calculator

The interactive mode is convenient for testing single line code and getting instant results. However, it has a limitation. We cannot save the code for later use. So, if we want to run the same code again, we need to retype it. This limitation can be overcome by the use of script mode.

(b) Script Mode

In the script mode, a Python program can be written in a file. This file can then be saved and executed. It can be edited or modified as per the requirements. The file made in the script mode is by default saved in the current working directory (eg. /home/linux/) and the extension to save a python file is ".py".

To use Python interpreter in the script mode, create a new file by executing the command sequence **File → New File** as shown in Figure 4.7. The same can be performed using the short-cut key **Ctrl+N**.

A new file will be opened, where we can write the Python statements. Figure 4.8 shows a file that contains three Python instructions. Once the coding is completed, save it by giving a suitable name with **.py** extension. Observe that the file is saved as **test.py** in this case.

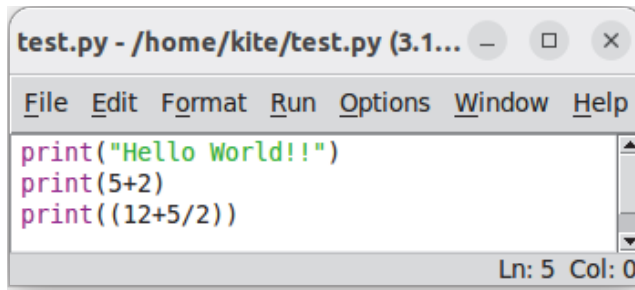


Figure 4.8: File containing Python statements

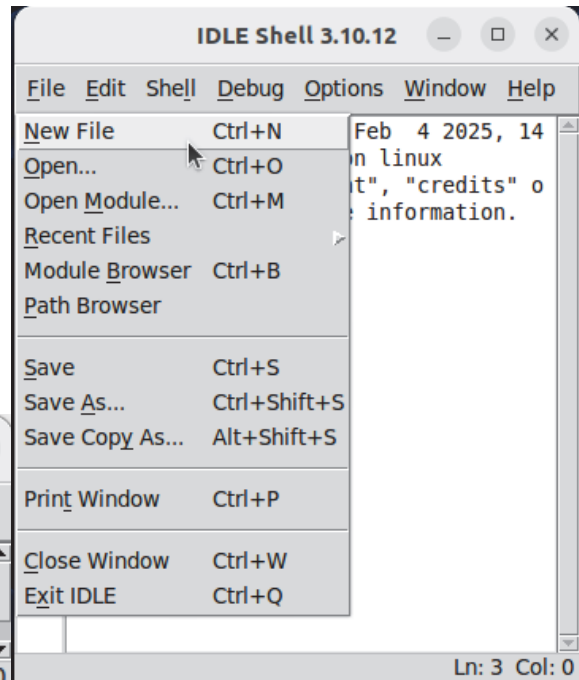


Figure 4.7: Creation of a file to use script mode in Python interpreter

Now, we can execute the program using the command sequence **Run → Run Module**, as shown in Figure 4.9.

The same can also be performed by pressing the function key **F5**. The Python shell will provide the output of the program as shown in Figure 4.10.

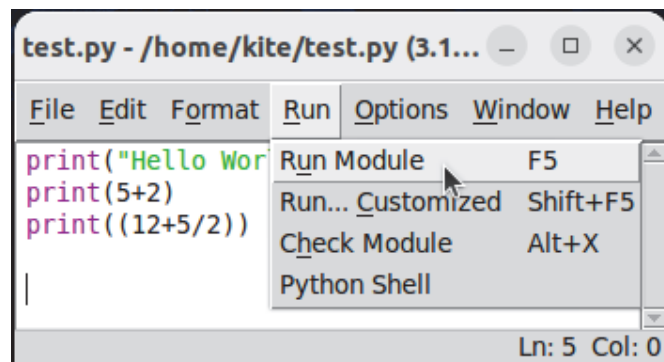


Figure 4.9: Procedure to execute Python program

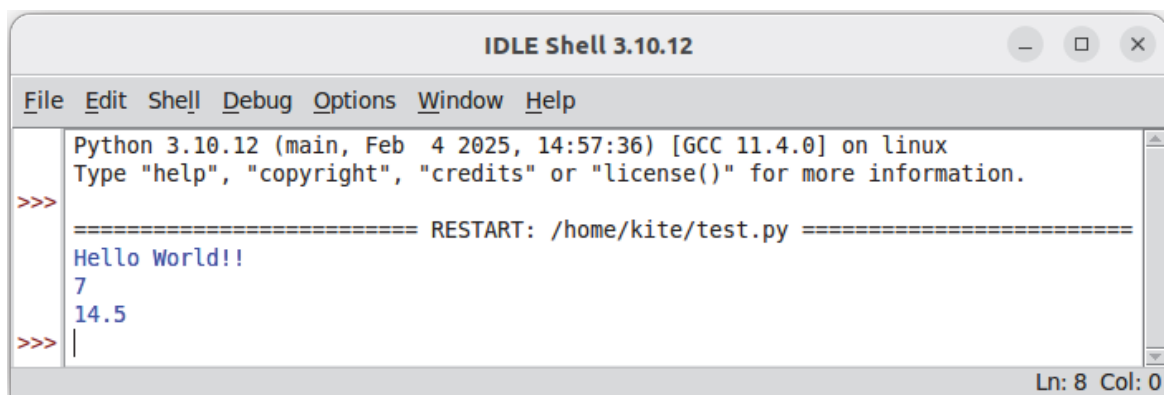


Figure 4.10: Output of Python program in the shell



Let us do

1. Open any online Python interpreter and write the following instructions and execute (run) them.
 - (i) `print ("Welcome to Python programming")`
 - (ii) `print (25 + 12)`
 - (iii) `print ("Hello" + "World")`
2. Based on the outputs of the above instructions, write down your observations on `print()` function.
3. Based on the outputs of instructions (ii) and (iii) in the first activity, compare the working of `+` operator.
4. Write down the instructions to solve the following problems, and execute them in a Python interpreter.
 - (i) Find the area of a circle with diameter 12 cm.
 - (ii) Find the perimeter of a rectangle with length of 7 cm and breadth of 5 cm.

In this chapter, we began our journey into programming by getting introduced to Python – a simple, powerful, and widely used programming language. Building on the earlier chapter, where we learned the basic ideas of problem solving and flowchart symbols, we have explored how these ideas are implemented in Python. We have become familiar with the Python IDLE environment and understood how to write and execute basic Python programs. This foundation prepares us for the next chapter, *Python Fundamentals*, where we will study Python's core concepts in detail, and begin developing more structured and meaningful programs.

Python Fundamentals

In this chapter, we will discuss the building blocks of Python program. These include character set, tokens, variables, data types, operators, expressions and input-output functions.

5.1 Character Set

A character set refers to the collection of valid characters that a programming language can recognise and use for writing programs. Every instruction or statement in Python must be composed of these characters. The Python character set includes the following:

- ◆ **Letters:** All uppercase letters (A–Z) and lowercase letters (a–z) of English language.
- ◆ **Digits:** All numeric digits (0–9).
- ◆ **Special symbols:** A wide range of symbols such as " ' ; : ! ~ @ # \$ % ^ & * () _ + - = { } [] \ . `
- ◆ **Whitespace characters:** Blank spaces, tabs, newline and carriage return.
- ◆ **Other characters:** All other ASCII and Unicode characters supported by Python.

5.2 Tokens

In natural languages, a statement consists of a group of words, where each word is a collection of letters. Similarly, a Python instruction consists of tokens, where each token is a collection of one or more characters. That is, a token is the smallest individual unit in a Python program. Python has five types of tokens – Keywords, Identifiers, Literals, Operators and Punctuators.

5.2.1 Keywords

A keyword is a special reserved word in Python that has predefined meaning and purpose. Each keyword is recognised by the Python interpreter and is used only for the purpose for which it has been defined. Python is a case-sensitive language and hence all keywords must be written exactly as specified. For example, True and true are treated differently in Python, where only True is a valid keyword. The list of Python keywords is shown in Table 5.1.

Table 5.1: Python keywords

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

In Python 3.10 and later versions, certain words called soft keywords are reserved only in particular situations. For example, `match`, `case`, and `_` (underscore) act as keywords in specific contexts. Being soft keywords, they can also be used as valid identifier names in other situations. We will discuss the soft keywords `match` and `case` in Chapter 6.

5.2.2 Identifiers

Identifiers are the user-defined words that are used to name different program elements such as variables, functions, objects, classes etc. While constructing identifiers certain rules are to be strictly followed for sustaining their validity in the program. An identifier is an arbitrary long sequence of letters, digits and underscores (`_`) that satisfies the following conditions.

- ♦ The first character must be a letter or underscore.
- ♦ White space and special characters are not allowed.
- ♦ Keywords cannot be used as identifiers.

Though there is no limit to the number of characters in an identifier, it is preferably kept short and meaningful. Examples of some valid identifiers are `count`, `avg_marks`, `FOR`, `_1stRank`.



Let us do

Choose the invalid identifiers from the following list of tokens and substantiate your findings.

<code>Data_rec</code>	<code>Num#</code>	<code>2ab</code>	<code>try</code>	<code>My File</code>
<code>_num1</code>	<code>Sum2Num</code>	<code>false</code>	<code>True</code>	<code>File.py</code>

5.2.3 Literals

We know that π and the acceleration due to gravity ' g ' are constants. In a similar way, a token that indicates a constant value is called a literal. Literals are often referred to as constants. Python supports different types of literals such as string literals, numeric literals, Boolean literals, and a special literal **None**. Python also allows grouping of literals, called literal collections such as list, tuple, dictionary and set.

1. **String literals:** A sequence of Python characters enclosed within a pair of single/double/triple quotes is called string literal. `"Computer Science"`, `'Python'` and `'''#number2'''` are

some examples. The unique feature of triple quotes is that they can be used to represent multiline strings as given below.

```
"""Python makes coding simple!
```

```
    Keep learning, keep growing."""
```

There are some specific characters, which represent certain non-graphic symbols such as tab and newline, and the punctuation marks like single quote (') and double quote ("). These characters begin with \ symbol followed by some specific characters as shown in Table 5.2. They are called escape sequences.

Table 5.2: Escape sequences

Escape sequence	Character
'\t' or "\t"	Tab
'\n' or "\n"	New line
'\'' or "'"	Single quote
'\"' or "\""	Double quote
'\\' or "\\"	Backslash

2. **Numeric literals:** These are constants in the form of numbers. They consist of integer literals, float literals and complex literals. **Integer literals** include both positive and negative numbers constituted by the digits 0 to 9. They can also include binary (eg. 0b1111 or 0B1111), decimal (15), octal (0o17 or 0O17) and hexadecimal (0xF or 0XF) literals. **Float literals** consist of both positive and negative real numbers having fractional parts. These consist of the digits and a period symbol (.). **Complex literals** are two-part numbers of the form $a+bj$, where a is known as the real part and b as the imaginary part, both a and b may be integer or float literals. The letter j stands for $\sqrt{-1}$. For example, -5 and 76 are integer literals; -3.14 and 0.5 are float literals; and $5+3j$ and $2-5j$ are complex literals.
3. **Boolean literals:** These are constants that represent truth values in Python. There are two Boolean literals, namely **True** and **False**. These are used to represent logical conditions and the result of comparison and logical operations.
4. **Special literal:** Python has a special literal **None**, which is used to denote nothing, no value or absence of value.

For each literal defined in a Python program, the Python interpreter internally allocates a space in memory to store the literal. This memory is often referred to as the **object associated with the literal**.

Literal Collections

In Python, literal collections are built-in ways of grouping multiple literals into a single unit. Literal collections can be lists, tuples, sets or dictionaries. These collections help us to organise data in various ways.

List is a collection of comma-separated elements within square brackets. The values can be of any type and can be changed as well. For example, `my_list = [25, "Python", 95.5, 'A+']` is a list.

Tuple is also a list of comma-separated values or elements within parentheses (round brackets). These values can also be of any type but they cannot be changed. For example, `my_tup = (23, "Computer Science", 'A', 35.8)` is a tuple.

Dictionary is an unordered set of key-value pairs within curly braces. The keys and values can be any type of data. In the example, `my_dict = {1: 'one', 2: 'two', 3: 'three'}`, 1, 2 and 3 are the keys; and 'one', 'two' and 'three' are the values.

Set is an unordered collection of elements in curly braces. Any type of data can be included in a set, but it does not allow duplicates unlike list and tuple. For example, `my_set = {'sat', 'sun', 20, 3.14}` is a set.

Here, ordered collection means the elements are stored in a specific sequence, and each item has a fixed position or index. So, the order in which you add elements is preserved. Unordered collection means the elements do not have a fixed order or position. Their arrangement may change, and you cannot access items using index numbers.

We will discuss literal collections in detail in the later chapters of this textbook.



Let us do

1. Some literals in Python are given below. Fill in the following table using these literals.

```
12          'Num#'          '''2ab'''          25.6          -5          "123"
'+='        '0'      {1, 'Ardra', 95}          (2, "Niketh", 'Science')
{"roll":25, "name":"Sneha", "score":85}          ["Nived", 20, 'A']
```

String	Character	Integer	Float	List	Tuple	Dictionary	Set

2. List out different types of Literal Collections and give example for each.

5.2.4 Operators

An operator is a token or a symbol that tells the interpreter about a specific operation. Python has a rich collection of operators. The operator is applied on a set of data called operands. The symbols `+`, `*`, `/`, `>`, `!=`, etc. are examples of operators. Various aspects of operators will be detailed in section 5.5 of this chapter.

5.2.5 Punctuators

In human languages, punctuation marks are used to structure sentences. Similarly, in Python, some special symbols that help organise the structure of programs are known as punctuators. The symbols `[`, `@`, `{`, `)` are some examples.

5.3 Variables

In a Python program, every value – whether it appears directly as a literal or is produced by evaluating an expression – must be stored somewhere in the computer’s memory. The Python interpreter creates this required memory during program execution whenever it is needed, and this memory location is called an object. Python allows us to create names, called identifiers, to refer to the objects in memory. These identifiers are known as variables. For example, the Python statement `x = 3` associates a variable `x` to an object corresponding to the integer literal 3.

Therefore, a variable is an identifier to which the three aspects of an object (see Figure 5.1) are associated with – the identity (the address of the memory location), the data type (type of the value stored in the location) and the value (the content in the location). We will see more about data types in the next section.

Any valid identifier can be used as the name of a variable. However, while naming variables, it is better to choose names that clearly describe their purpose or the type of data they store (Eg. `totalMarks`, `student_name`, `AvgScore`).

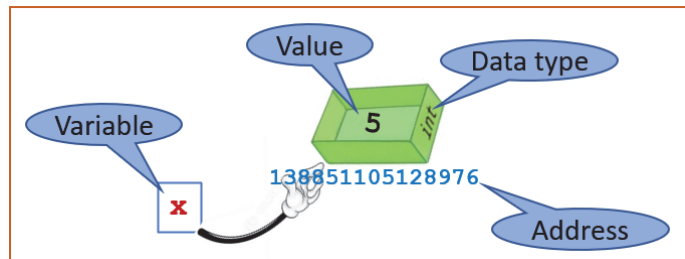


Figure 5.1: An object referenced by a variable

5.4 Data Types

Every literal that we discussed in section 5.2.3 has an associated data type. A data type refers to a set of literals or values, together with the operations defined on these literals. For instance, the integer data type refers to the set of all integers, with the standard arithmetic operations allowed on them. The Boolean data type is defined by the set `{True, False}`, together with the operations `and`, `or` and `not`. The string data type is defined by the set of all strings along with operations such as string concatenation. The float type consists of the set of all floating-point numbers together with the operations such as addition, subtraction, multiplication, division etc. Note that the same operator may have different meanings for different data types. For example, the `+` symbol refers to addition for integer type, whereas `+` refers to the concatenation operation for strings.

In addition to simple data types like integers and strings, Python supports grouped data items such as lists, tuples and dictionaries. Python treats both simple data types as well as their aggregates as classes, following the terminology used in Object Oriented Programming. You will learn more about **Object Oriented Programming** in

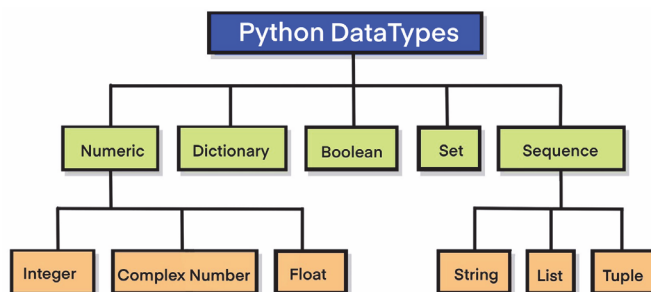
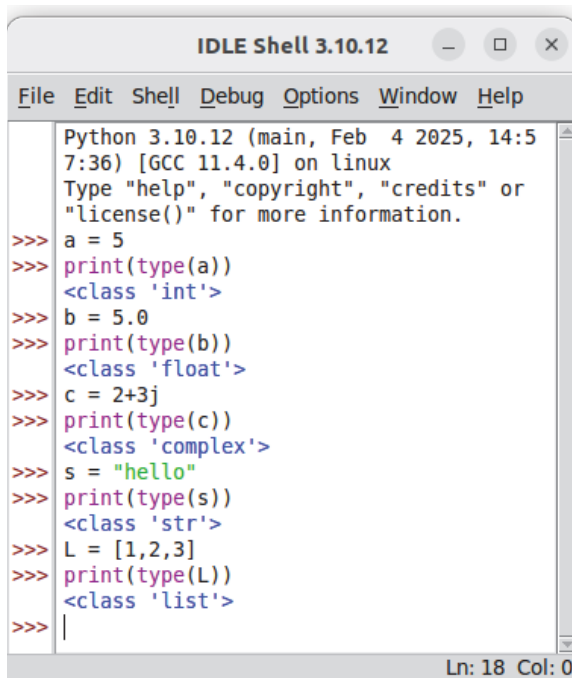


Figure 5.2: Python data types

Class XII. A hierarchy of simple and aggregate data types supported by Python is shown in Figure 5.2.

The class names given by Python to these data types are given below.

- Numeric : int, float, complex
- Sequence : str, list, tuple
- Dictionary : dict
- Boolean : bool
- Set : set, frozenset



```

Python 3.10.12 (main, Feb 4 2025, 14:5
7:36) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or
"license()" for more information.
>>> a = 5
>>> print(type(a))
<class 'int'>
>>> b = 5.0
>>> print(type(b))
<class 'float'>
>>> c = 2+3j
>>> print(type(c))
<class 'complex'>
>>> s = "hello"
>>> print(type(s))
<class 'str'>
>>> L = [1,2,3]
>>> print(type(L))
<class 'list'>
>>>
  
```

Figure 5.3: Demonstration of Python data types

The statements shown in Figure 5.3 demonstrate how Python refers to the data types.

In Figure 5.3, the function **type()** finds the data type of the value given in parentheses and the function **print()** shows the result in the Python shell. These are built-in functions of Python. We shall have a detailed discussion on the concept of functions in Chapter 7. From the output, we can infer the fact that the variable **a** refers to an object of class **int**, the variable **b** refers to an object of class **float** and the last variable **L** refers to an object of class **list**.

In some languages, data types are used to declare variables. But Python variables do not require explicit declaration. In Python,

assigning value to a variable results in the creation of an object with the given value, and the variable holds a reference to that object. The data type of the variable is the same as the created object. The data type of the variable will be changed when it is assigned

a value of a different data type. Consider the following Python statements:

```

num = 10
msg = "Hello"
num = 3.5
  
```

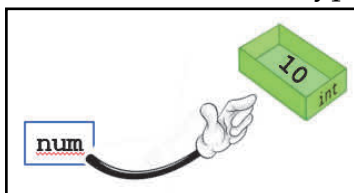


Figure 5.4(a): After the execution of `num = 10`

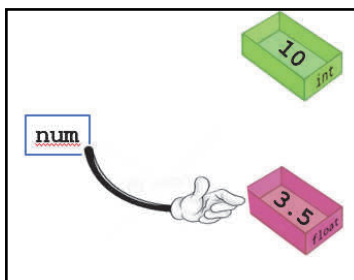


Figure 5.4(b): After the execution of `num = 3.5`

The python interpreter will associate the variable `num` with an object in memory having the literal value 10 when the statement `num = 10` is executed (see Figure 5.4(a)). Hence the type of `num` will be `int`. The statement `msg = "Hello"` will set the variable `msg` to refer to an object that contains the string literal "Hello" and hence `msg` will be of `str` type.

After the execution of `num = 3.5`, the variable `num` will refer to an object of value 3.5, thus changing its type to `float` (see Figure 5.4(b)).

5.4.1 Mutable and Immutable Data Types

When we assign a value to a variable in a program, the Python interpreter allocates a memory object to store that value. If we later change the value of the same variable, Python must update the value stored in the object that the variable refers to. The way the update occurs depends on whether the type of the object referred to by the variable is mutable or immutable. Data types that allow modification of their values after creation are called **mutable**, while those which do not allow any change after they are once created are called **immutable**. In Python, lists, dictionaries and sets are mutable types, while `int`, `float`, `str` and `tuple` are immutable types.

If a variable referring to an object of mutable type is updated, Python changes the value inside the original memory object, and the variable continues to refer to the same object. But when we update a variable referring to an object of immutable type, Python does not change the existing object. Instead, it creates a new memory object with the updated value, and the variable is made to refer to this new object. The old object cannot be referenced by this variable any more (refer to Figure 5.4(a) and 5.4(b)). But it is possible to refer to this object by some other variable which has already been referring to it earlier.

Consider the Python statements `num=10` and `temp=num`. Here the variable `num` is assigned the value 10, and hence `num` will refer to an object that contains the integer value 10 (refer to Figure 5.5(a)). Since the variable `temp` is assigned the value of `num`, `temp` will also refer to the same object (refer to Figure 5.5(b)). However, when the statement `num=num*num` is executed, the Python interpreter creates a new object in memory and stores the value 100 (`num*num`) in it. Now onwards, the variable `num` will refer to the new object, and not the old (refer to Figure 5.5(c)). This is because the variable `num` refers to the object of type `int`, which is an immutable data type. But the variable `temp` still refers to the object that contains the value 10.

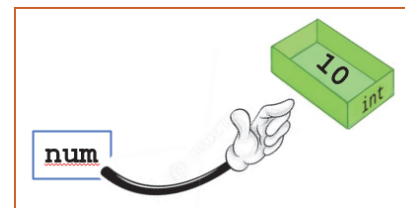


Figure 5.5(a): After the execution of `num = 10`

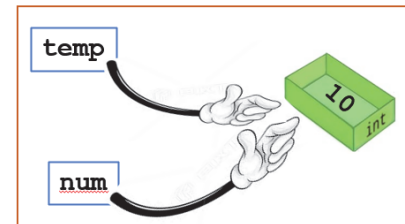


Figure 5.5(b): After the execution of `temp = num`

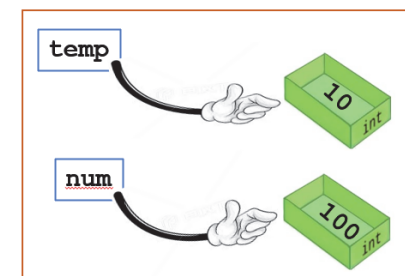


Figure 5.5(c): After the execution of `num = num * num`



Let us do

- Observe the following Python statements and identify the data type of the variables.

(a) <code>n=25</code>	(b) <code>PI=3.14</code>	(c) <code>name="Guido"</code>	(d) <code>flag=True</code>
-----------------------	--------------------------	-------------------------------	----------------------------
- Verify your answer by using a Python interpreter.
- Identify the data types of the following:

(a) 234	(b) 0.54	(c) "True"	(d) False	(e) (1, 45.65, "Raihan")
(f) {3, "Neelima", 'F'}	(g) [5, "Joel", "M"]			

5.5 Operators

In Python, operators are special symbols or keywords that are used to perform operations on values and variables. They act upon the given data and produce a result. The data which are the participants in the operation are known as **operands**. For example, in the Python instruction $5 + 3$, the symbol $+$ is an operator that adds two numbers 5 and 3, which are the operands. Operators in Python can be used not only with numbers, but also with strings, lists, and other data types. Depending on the type of operation, Python offers various categories of operators including arithmetic, relational, logical, assignment, identity and membership operators.

Operators that require only one operand are called **unary operators** and those which require two operands are called **binary operators**.

Understanding operators is essential, as they serve as the building blocks of expressions and enable us to write meaningful programs.

5.5.1 Arithmetic Operators

Python provides arithmetic operators that are used not only for the four basic operations – addition, subtraction, multiplication and division, but also for performing modulus (remainder), floor division, and exponentiation (power) operations. While using arithmetic operators, we should give utmost care to the data type of the operands, as they influence the data type of the result. When arithmetic operators are used, the operands may be of `int`, `float` or `complex` type, influencing the type of the result as detailed below.

- ◆ If both the operands are of integer type, the result of the arithmetic operations (addition, subtraction, multiplication, floor division, modulo) will also be of integer type.
- ◆ If both operands are floating-point numbers, the result will be a floating-point number.
- ◆ When an operation involves both an integer and a floating-point number, Python automatically promotes the integer to a floating-point number before performing the operation. The result will then be a floating-point number.
- ◆ The standard division operator (`/`) always returns a floating-point number, regardless of whether the operands are integers or floats.

Table 5.3 illustrates the above scenario using the addition operator `+`. The same principle applies to all other arithmetic operators.

Table 5.3: Impact of data type on arithmetic operator +

Type of Operand 1	Type of Operand 2	Type of the Result	Example
int	int	int	>>> 7 + 5 12
int	float	float	>>> 7 + 2.5 9.5
float	int	float	>>> 7.5 + 5 12.5
float	float	float	>>> 7.5 + 2.5 10.0



Let us do

1. Prepare a table similar to Table 5.3 and include the data type complex as an operand and predict the type of the result for all combinations.
2. Verify your prediction with the help of Python interpreter using your examples in each combination.

Table 5.4 lists the arithmetic operators in Python and describes each of their operations with the help of examples. Assume that the two variables **num1** and **num2** are assigned values as follows.

```
>>> num1 = 5
>>> num2 = 2
```

Table 5.4: Arithmetic operators in Python

Operator	Operations	Description	Example
+	Addition & Concatenation	Adds the two numeric values on either side of the operator. Concatenates (joins) two strings on its either side.	>>> num1 + num2 7 >>> "Hello" + "Guys" HelloGuys
-	Subtraction	Subtracts the operand on the right from the operand on the left.	>>> num1 - num2 3
*	Multiplication & Repetition	Multiplies the two values on both sides of the operator. Repeats the item on the left of the operator if the first operand is a string and the second operand is an integer.	>>> num1 * num2 10 >>> "Hello" * 2 HelloHello
/	Division	Divides the operand on the left by the operand on the right and returns the result.	>>> num1 / num2 2.5

%	Modulus	Divides the operand on the left by the operand on the right and returns the remainder.	>>> num1 % num2 1
//	Floor Division	Divides the operand on the left by the operand on the right and returns an integer value by rounding down the nearest integer value.	>>> num1 // num2 2 >>> 5.0 // 2.0 2.0
**	Exponent	Performs exponential (power) calculation on the operands - that is, raises the operand on the left to the power of the operand on the right.	>>> num1 ** num2 25

From Table 5.4, it is clear that the above operators are binary as they require two operands to operate upon. But it is to be noted that + and - are used as unary operators also. In that case, the number will be expressed as positive or negative. In such situations, the symbols + and - are called unary plus and unary minus respectively.

It should also be noted that when the operator + is used with string operands, it acts as concatenation operator which joins the two strings. The + operator is also used with lists and tuples to join them. We will see more about these operations in later chapters. Similarly, in the case of * operator, when the first operand is a string and the second is a positive integer, the string will be concatenated with itself the specified number of times. We will discuss these operations in Chapter 8.

5.5.2 Relational Operators

A relational operator is used to compare the values of two operands and checks the relationship between them, such as greater than, less than and equal to. All the relational operators are binary and hence they require two operands. The output of these operations will be a Boolean value, i.e., either **True** or **False**. Table 5.5 shows the relational operators and their operations. Consider the Python variables shown below for illustration.

```
>>> num1 = 5
>>> num2 = 2
>>> num3 = 5
>>> str1 = "Good"
>>> str2 = "Bad"
```


Table 5.5: Relational operators in Python

Operator	Operations	Description	Example
<code>==</code>	Equal to	If the values of two operands are equal, the condition evaluates to True, otherwise it is False.	<pre>>>> num1 == num2 False >>> str1 == str2 False >>> str1 == "Good" True</pre>
<code>!=</code>	Not equal to	If the values of two operands are not equal, the condition results in True, otherwise it is False.	<pre>>>> num2 != 2 False >>> str1 != str2 True</pre>
<code>></code>	Greater than	If the value of the left-side operand is greater than the value of the right-side operand, the condition evaluates to True, otherwise it is False.	<pre>>>> num1 > num3 False >>> str1 > str2 True</pre>
<code>>=</code>	Greater than or equal to	If the value of the left-side operand is greater than or equal to the value of the right-side operand, the condition is evaluated to True, otherwise it is False.	<pre>>>> num2 >= num3 False >>> num1 >= num3 True >>> str1 >= str2 True</pre>
<code><</code>	Less than	If the value of the left-side operand is less than the value of the right-side operand, the condition results in True, otherwise it is False.	<pre>>>> num1 < num2 False >>> str1 < "good" True</pre>
<code><=</code>	Less than or equal to	If the value of the left operand is less than or equal to the value of the right operand, the condition results in True, otherwise it is False.	<pre>>>> num1 <= num2 False >>> num1 <= num3 True >>> "Good" <= str1 True</pre>

Table 5.5 shows that while comparing strings, their alphabetical precedence is considered. Note that `"Good" < "good"` returns True since the Unicode of uppercase letters is smaller than that of the lowercase letters. As we discussed in Chapter 2, Unicode is a character encoding system that represents letters, digits, and symbols using numeric codes interpreted by computers. Another point to be noted is that if one of the operands for relational operators other than `==` and `!=` is numeric and the other is a string, Python interpreter will report a `Type Error`.

If the two operands of an operator are of incompatible types, Python attempts to convert one of the operands so that the operation becomes type compatible. For instance, the operation `3.0 < 4` evaluates to True, because 4 is implicitly converted to float type and compared with 3.0. But, `3 < "3"` reports a `Type Error`.

5.5.3 Logical Operators

Python has three logical operators namely **and**, **or**, **not**. Among these, **not** is unary and the others are binary. Normally, a logical operator expects the operands to be of Boolean type and the operation results in a Boolean value.

In the case of **and** operator, if the first operand is **True**, the output will be the second operand. If the first operand is **False**, the output will be the first operand itself. In the case of **or** operator, if the first operand is **True**, the output will be the first operand itself. If the first operand is **False**, the output will be the second operand. This concept is applicable in the case of other data types also. In this case, a value of any data type may be considered as **True** or **False**. The values such as **None**, **False**, 0 (zero), empty string (""), empty tuple (()), empty list ([]), empty dictionary ({}), and a few other empty

values are treated as **False** and all other values as **True**. For example, 3 and 0 evaluates to 0. The operation "Hello" or "" will be evaluated to "Hello".

The above concept is shown in Table 5.6. Here X represents any data used as the second operand for the specified operation

Table 5.6: Operations of **and**, **or** operators

Operation	First Operand	Second Operand	Result
and	True	X	X
	False	X	False
or	True	X	True
	False	X	X

Consider the following Python statements for the illustration of Logical operators as shown in Table 5.7.

```
>>> num1 = 15
>>> num2 = 12
>>> num3 = 5
```

Table 5.7: Logical operators in Python

Operator	Operations	Description	Example
and	Logical AND	If both the operands are True, then the output becomes True.	<pre>>>> True and True True >>> (num1>num2)and(num1>num3) True >>> (num1<num2)and(num1<0) False</pre>
or	Logical OR	If any of the two operands is True, then the output becomes True.	<pre>>>> (num1<num2)or(num1>0) True >>> False or False False</pre>
not	Logical NOT	It is used to reverse the logical state of its operand.	<pre>>>> not (num1>num2) False >>> not False True</pre>

5.5.4 Assignment Operators

The assignment operator (=) is used to associate a variable to a memory object that contains a value. The variable on the left side of = will refer to the object in which the value of the expression on the right side is stored. The following statements are examples for such assignments.

```
>>> num1 = 15
>>> num2 = num1 + 12
>>> str1 = 'hello'
>>> str1 = str1 + ' friends'
```

After the execution of the first statement, the variable **num1** will refer to an object in the memory that contains the integer literal 15 and hence type of **num1** becomes **int**. When the second statement is executed, the value stored in the object referred to by **num1** (i.e., 15) and the value 12 are added and the result 27 is stored in a new object in the memory. This memory object is referred to by the variable **num2** which makes it **int** type.

Similarly, the variable **str1** refers to the object that contains the string literal 'hello'. But when the last statement is executed, the value referred to by **str1** ('hello') and the string literal 'friends' are concatenated, and the string 'hello friends' will be stored in a new object in the memory. Note that a new object is created since **str1** is an immutable data type. After the assignment, **str1** now refers to this new object. Since no other variable refers to the literal 'hello', the object has no reference.

Python also provides **compound assignment operators** that combine assignment with arithmetic operators.

The statement: `str1 = str1 + ' friends'`
can be written as: `str1 += ' friends'`

Table 5.8 shows the compound assignment operators and the equivalent arithmetic assignment operation. Assume that **a** and **b** are variables. It is applicable only when the operand on the left side of = and one of the operands on the right side of = are the same.

Table 5.8: Compound assignment operators in Python

Compound Assignment Operator	Equivalent arithmetic assignment operation
<code>a += b</code>	<code>a = a + b</code>
<code>a -= b</code>	<code>a = a - b</code>
<code>a *= b</code>	<code>a = a * b</code>
<code>a /= b</code>	<code>a = a / b</code>
<code>a %= b</code>	<code>a = a % b</code>
<code>a //= b</code>	<code>a = a // b</code>
<code>a **= b</code>	<code>a = a ** b</code>

5.5.5 Identity Operators

Identity operators check whether two variables refer to the same object. They are also used to find out the type of a variable. In Python, there are two identity operators - **is** and **is not**. Table 5.9 describes their mode of operation.

Table 5.9: Identity operators in Python

Operator	Description	Example
<code>is</code>	Evaluates to True if the variables on either side of the operator refer to the same memory object and False otherwise. <code>var1 is var2</code> is True, if <code>id(var1)</code> is equal to <code>id(var2)</code> .	<pre>>>> num1 = 5 >>> num2 = num1 >>> num1 is num2 True >>> type (num1) is int True</pre>
<code>is not</code>	Evaluates to False if the variables on either side of the operator refer to the same memory object and True otherwise. <code>var1 is not var2</code> is True if, <code>id(var1)</code> is not equal to <code>id(var2)</code> .	<pre>>>> num1 = 5 >>> num2 = num1 >>> num1 is not num2 False >>> type (num1) is not int False</pre>

The function `id()` used in Table 5.9 is a built-in function that gives the memory address of an object. We will discuss the concept of built-in functions in Chapter 7.

5.5.6 Membership Operators

We have seen that sequence type data consists of string, list and tuple. Membership operators are used to check if a value is a member of the given sequence or not. Table 5.10 shows these operators and their operations.

Table 5.10: Membership operators in Python

Operator	Description	Example
<code>in</code>	Evaluates to True if a specified variable/value is found in the given sequence and False otherwise.	<pre>>>> a = [10, 15, 100] >>> 10 in a True >>> '15' in a False</pre>
<code>not in</code>	Evaluates to True if a given variable/value is not found in the specified sequence and False otherwise.	<pre>>>> a = [10, 15, 100] >>> 10 not in a False >>> '15' not in a True</pre>

5.6 Expressions

An expression in Python is a combination of operands (such as variables, constants or values) and operators that is evaluated to produce a result. For example, `5+3`, `x/y` and `a>b` are all expressions. When an expression is executed, Python evaluates it and creates an object to store this result. This result is also known as the return-value of the expression. This value can be of any data type such as integer, float, string or Boolean. The data type of the object containing this value will be determined by the data type of the operands, as we discussed in the previous section. A value or a

standalone variable is also considered as an expression since it returns a value, but an operator alone is not an expression.

5.6.1 Precedence of Operators

When an expression has different types of operators, Python follows a rule called operator precedence to decide which operator to evaluate first. An operator with higher precedence is evaluated before the one with lower precedence. Most of the operators we have discussed so far are binary operators, which means they work with two operands. Some operators, called unary operators, work with only one operand and usually have a higher precedence than binary operators. Table 5.11 shows the precedence of operators used in Python.

Table 5.11: Precedence of Python operators

Order of precedence	Operators	Description
1	**	Exponentiation (raise to the power)
2	+ -	Unary plus, unary minus
3	* / % //	Multiplication, division, modulus, floor division
4	+ -	Addition, subtraction
5	< <= > >= == !=	Relational or comparison operators
6	= %= /= //= -= += *= **=	Assignment and compound assignment operators
7	is is not	Identity operators
8	in not in	Membership operators
9	not	Logical NOT
10	and	Logical AND
11	or	Logical OR

Parentheses play a key role in determining the order of evaluation in an expression. Normally, operators are evaluated according to their precedence, but when parentheses are used, the operations enclosed within them are evaluated first, regardless of the usual precedence rules. This helps make expressions clearer and reduces confusion. When two or more operators in an expression have the same precedence, Python evaluates them from left to right.

For example, consider the expression `5 + 7 - 2 * 4`. The order of evaluation will be as follows:

- Step 1 : $2 * 4$ results in 8 and now $5 + 7 - 8$ needs to be evaluated.
 Step 2 : $5 + 7$ results in 12 and $12 - 8$ remains for evaluation.
 Step 3 : $12 - 8$ will be evaluated to 4, which will be the final result.

On the other hand, if the expression was $5 + (7 - 2) * 4$, then the result would have been 25 as detailed below.

- Step 1 : $7 - 2$ results in 5 and now $5 + 5 * 4$ is to be evaluated.
 Step 2 : $5 * 4$ results in 20 and $5 + 20$ remains for evaluation.
 Step 3 : $5 + 20$ will be evaluated to 25 and this will be the value returned by the expression.



Let us do

- Find out the value returned by and its type in each of the following expressions assuming the variables $a=5$, $b=10$, $c=2.5$, $d="hello"$.

(i) a / b	(vi) $(a + b) \% 2$	(xi) $(a/b) > c$
(ii) $a + b - c$	(vii) $a ** 3 - b * c$	(xii) $('h'+'a') * 3$
(iii) $d * 3$	(viii) $a > b$	(xiii) $"Hi" < "Hello"$
(iv) $d + "friends"$	(ix) $(a>b) \text{ and } (b>c)$	(xiv) $'A' == 'a'$
(v) $b // 3$	(x) $(a\%10) > 0$	(xv) $a * (b - c) / 2$
- What will be the value of `ans` in each of the following cases, if `ans=25` initially?

(a) <code>ans += 10</code>	(b) <code>ans /= 10</code>	(c) <code>ans %= 10</code>	(d) <code>ans -= ans*5</code>
----------------------------	----------------------------	----------------------------	-------------------------------
- Evaluate each of the expressions/statements given above using Python interpreter. Print the type of each output value with the help of `type()` function.

5.7 Statements

A statement in Python is an instruction given to the computer to perform a specific task such as input, output, assignment or control program flow. Python provides facilities to specify these tasks in a program. Let us begin with the statements to perform assignment, input and output operation. The statements required to control the program flow will be discussed in Chapter 6. Python executes the program statements one by one in the order they appear or as per the control flow.

5.7.1 Assignment Statement

We have already seen some cases of assignment statements in section 5.5.4, where the assignment operator `=` is discussed. The following are some examples of assignment statements.

```
num = 125
msg = 'Hello World'
num = num / 10
```

Object references after the execution of the first two statements are given in Figure 5.6(a) and object references after the execution of the third statement is shown in Figure 5.6(b).



Figure 5.6(a): Object references before the execution of the 3rd statement

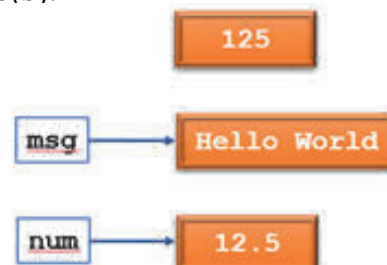


Figure 5.6(b): Object references after the execution of the 3rd statement

Python is very versatile with assignment operation. We can assign the same value to multiple variables in a single statement as: **a = b = c = 10**. Here, all the three variables a, b and c will refer to the same location that contains the value 10 as shown in Figure 5.7(a).

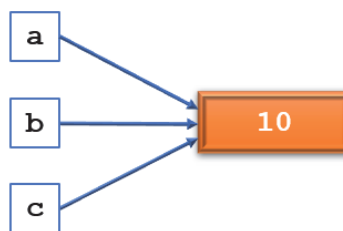


Figure 5.7(a): Object references for the multiple variable assignment

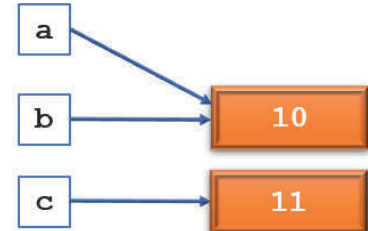


Figure 5.7(b): Object references after the update of c

If the statement **c+=1** is executed followed by the above assignment, the object reference will be as shown in Figure 5.7(b)

We can even assign multiple values to multiple variables in a single statement as: **a, b, c = 10, 15, 20**. It will assign the values in order, i.e., the first variable is given the first value, the second variable the second value and so on. That means, the variable a refers to the memory object containing 10, b refers to the memory object containing 15 and c refers to the memory object containing 20. In such cases, we can use expressions also instead of literals.

5.7.2 Input Statement

In many programs, it is necessary to take input data or information from the user so that the program can process it and produce the required output. Python provides the **input()** function for this purpose. When the **input()** function is used, it displays a prompt message asking the user to provide data. After the user types the input and presses the Enter key, the data is sent to the program. The syntax for **input()** is:

```
input([prompt])
```

where 'prompt' is the string to be displayed on the screen prior to take the input, and is optional.

If we want to store the entered data, a variable is to be used as follows.

```
var = input([prompt])
```

For example, on executing the statement

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

the string Enter your name: will be displayed on the screen followed by the cursor waiting for user input. Once we type a name, for example, Pavithra, and press Enter key, a memory object will be created and the input string will be stored, and this object will be referred to by the variable **name**.

It is important to note that the **input()** function always reads the entered data as a string, even if the user types a number. If the program needs to use this input as a number, it must be converted to an appropriate numeric data type such as **int** or **float** before performing any calculations. It can be achieved by using the built-in functions like **int()** and **float()**. The usage of **int()** function is illustrated through a program given in section 5.8.2.

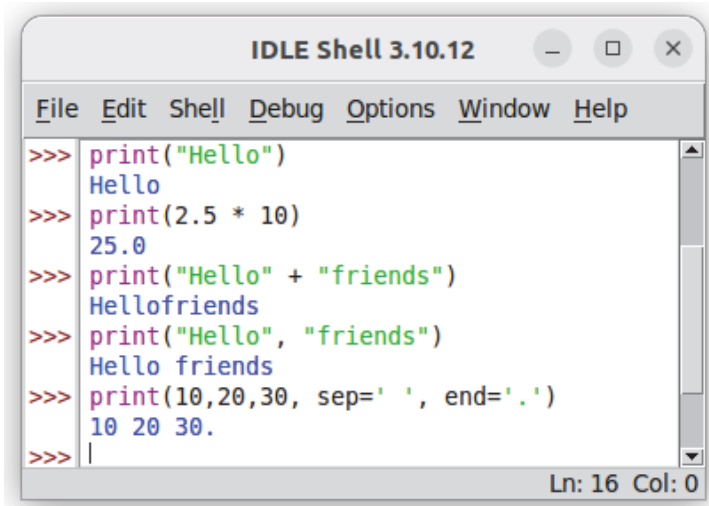
5.7.3 Output Statement

After processing data, a program often needs to display the results to the user. In Python, this is done using the **print()** function. The **print()** function sends information, such as text, numbers or the result of calculations, to the screen. Multiple values can be displayed together, and special characters like '**\n**' can be used to format the output. This allows the program to communicate results clearly and in a readable form, making it easier for the user to understand the output of the program. The syntax for **print()** is:

```
print(value [, ..., sep = char1 , end = char2])
```

where value is the data or information to be displayed, char1 and char2 are the characters to be used as the values of the parameters sep and end respectively. The sep is an optional parameter which specifies the separator between the output values. We can use a character, number or string to separate values while using the **print()** function. If no separator is specified, Python uses a space by default. Another optional feature is end, which allows us to add any string after the last value in the output. By default, Python moves to a new line after printing.

For example, Figure 5.8 shows various modes of usage of **print()** function and the output obtained.



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> print("Hello")
Hello
>>> print(2.5 * 10)
25.0
>>> print("Hello" + "friends")
Hellofriends
>>> print("Hello", "friends")
Hello friends
>>> print(10,20,30, sep=' ', end='.')
10 20 30.
>>> |
Ln: 16 Col: 0

```

Figure 5.8: Different usages of print() function

5.8 Python Programs

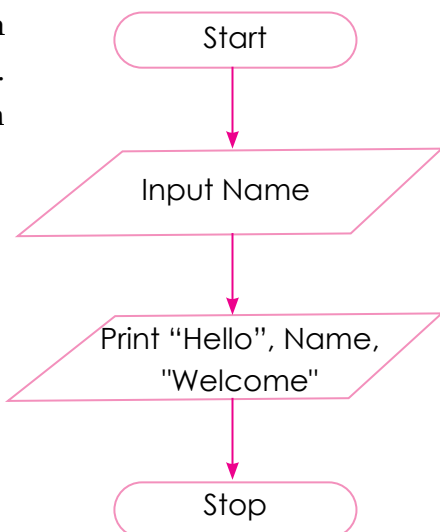
We have learned the basic concepts of Python, such as variables, data types, operators, expressions, statements and input/output functions. Now, we are ready to write our first simple program. A simple program in Python is a small set of instructions that performs a specific task, such as taking input from the user, processing it and displaying the output. In such programs, the instructions are executed **one by one** in the order they appear. This order of execution is called the **sequential flow**. Writing simple programs helps us understand how Python executes statements, evaluates expressions and uses functions like **input()** and **print()** to interact with the user. These programs form the foundation for developing more complex applications in the future.

5.8.1 Program to accept user's name as input and display a greeting

In Chapter 3, we discussed the concepts of algorithm and flowchart as the foundation for problem solving. So, let us develop an algorithm and construct a flowchart to solve this problem.

Algorithm to accept a name and print with a greeting

Step 1: Start
 Step 2: Input Name
 Step 3: Print "Hello", Name, "Welcome"
 Step 4: Stop



Program to accept a name and print with a greeting

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

Figure 5.9(a) shows the above Python program in the IDLE IDE. When you run this program, you will get the output as shown in Figure 5.9(b).

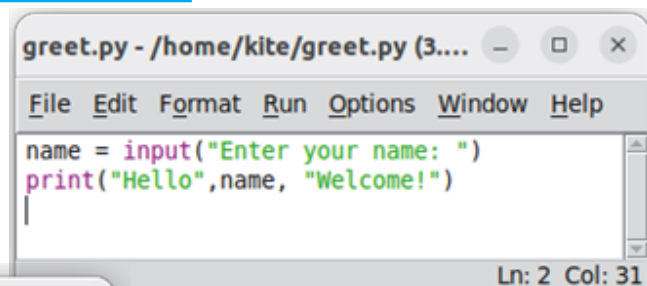


Figure 5.9(a): Python code of Program 5.8.1

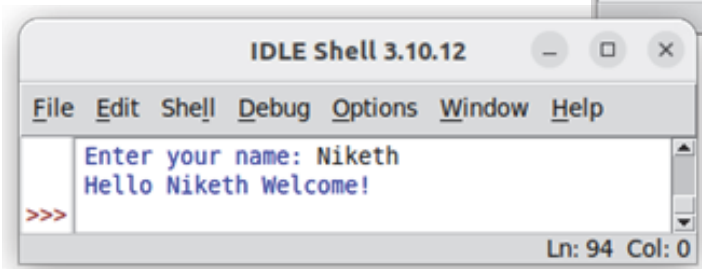
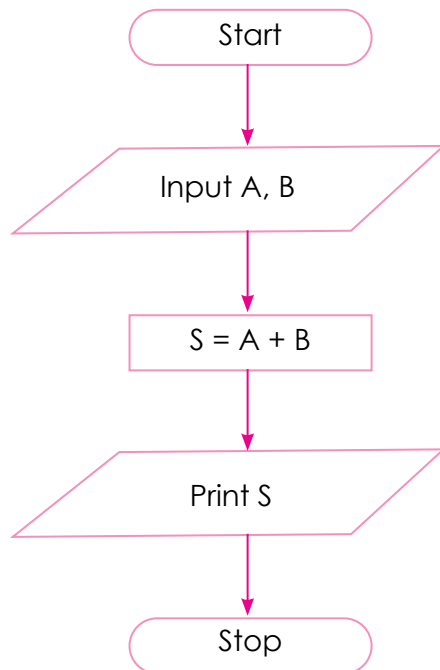


Figure 5.9(b): Output of Program 5.8.1

5.8.2 Program to find the sum of two numbers

In this problem the numbers are not provided, and hence the program should prompt the user to input two numbers.



Algorithm to find the sum of two numbers

Step 1: Start
 Step 2: Input A, B
 Step 3: $S = A + B$
 Step 4: Print S
 Step 5: Stop

Program to find the sum of two numbers

```
#accept two numbers
num1 = input("Enter first number: ")
num2 = input("Enter second number: ")
#calculate the sum
s = int (num1) + int (num2)
#print the result
print("Sum = ", s)
```

```
sum2num.py - /home/kite/sum2...
File Edit Format Run Options Window Help
#accept two numbers
num1 = input("Enter first number: ")
num2 = input("Enter second number: ")
#calculate the sum
s = int(num1) + int(num2)
#print the result
print("Sum =", s)
Ln: 9 Col: 0
```

Figure 5.10(a): Python code of Program 5.8.2

Observe the above algorithm and flow chart to understand how the logic of the solution is derived.

Figure 5.10(a) shows the Python code and 5.10(b) shows the output of this program in IDLE shell. You can see certain statements starting with # symbol. It will be discussed in the next section.

```
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Enter first number: 12
Enter second number: 43
Sum = 55
>>>
Ln: 9 Col: 0
```

Figure 5.10(b): Output of Program 5.8.2

An important feature in this program is the use of the `int()` function. As mentioned earlier (Section 5.7.2), the data received by

the **input()** function is always treated as a string, which cannot be used directly for arithmetic operations. The **int()** function converts a string of digits into an integer so that calculations like addition can be performed. If we want to work with decimal numbers, the **float()** function can be used instead of **int()**. This conversion can also be done while taking input, as shown in the statement:

```
num1 = int(input("Enter first number: "))
```

5.8.3 Comments in Programs

The program written in the previous section contains three lines starting with # symbol. They are **comments** in the program. Comments are lines in a program that the Python interpreter ignores and does not execute. They are used to explain the code or make it easier to understand for anyone reading it. In Python, a single-line comment begins with the # symbol. If you want to provide multi-line comment, it should be given within a pair of triple quotes (""" or '''). Comments help programmers document their code and make it more readable. It is a kind of documentaion process that takes place during the final phase of programming, as we learned in Chapter 3.



Let us do

Write an algorithm, construct a flowchart, and develop a program to solve each of the following problems.

1. To find the sum and the average of three numbers.
2. To find the area and the perimeter of a rectangle.
3. To input the diameter of a circle and find its area and perimeter.
4. To input an amount in paise and convert it into rupees and paise. (For example, if the input amount is 575 paise, the output should be Rs. 5 Ps. 75)
5. To input a time quantity in seconds and represent it in Hr : Min : Sec format. (For example, if the input is 7700 seconds, the output should be 2 Hrs : 8 Mins : 20 Secs)
6. Write a program to swap two numbers using a third variable.
7. Write a program to swap two numbers without using a third variable.
8. Write a Python program to convert temperature in degree Celsius to degree Fahrenheit. If water boils at 100 degree C and freezes as 0 degree C, use the program to find out the boiling point and freezing point of water on the Fahrenheit scale. (Hint: $T(^{\circ}\text{F}) = T(^{\circ}\text{C}) \times 9/5 + 32$).

In this chapter, we explored the fundamental building blocks of Python programming. We learned Python's character set and tokens, understood how variables, data types, and operators are used to process data, and became familiar with input and output functions for interacting with the user. We also wrote simple sequential programs that execute statements in the order as they appear, helping us apply these basic concepts to practice. With this strong foundation, we are now ready to move on to the next chapter *Control Statements*, where we will learn statements such as decision making statements and looping statements which allow us to write more flexible and powerful programs.

Control Statements

In the earlier chapters, we learned basic Python programs which contains sequential statements such as input, output and assignment. The instructions in those programs execute one after another in a simple, linear fashion.

However, real-life situations are not always like this. We often need to make choices, whether deciding a route based on traffic, selecting a course after Class 10, or determining eligibility for an exam. Similarly, a program may need to choose between alternative paths of execution based on conditions.

This is where control statements become essential. They allow a program to make decisions, repeat actions, or skip certain instructions. In this chapter, we discuss problem solving that involves selection, and iteration. We also introduce the Python statements that alter the default flow of execution, enabling programs to behave intelligently and efficiently.

6.1 Control Statements in Python

To go beyond sequential programming, Python provides control statements that interact with the computer's processing units – especially the Arithmetic and Logic Unit (ALU), which evaluates conditions and guides the program to take the correct path. These control statements help the program decide what to do next based on logical conditions.

When a program reaches a point where a decision must be made, it evaluates conditions using comparisons such as greater than, less than, or equal to. These comparisons are processed by the Arithmetic and Logic Unit (ALU), which returns either True or False. This logical result guides the program to execute, skip, or choose between different statements. Similar decision-making occurs in real life, for example, assigning grades based on marks scored: a student receives A+, A, B+, and so on only after comparing the marks with predefined grade boundaries.

In the same way, Python evaluates conditions to decide the program's next action. Sometimes, the decision affects only one step, but in other cases, it determines whether a task should be repeated. For instance, repeatedly adding subject marks until all subjects are processed is a looping situation where the repetition



continues as long as a condition remains true. Thus, both selection (decision making) and repetition (iteration) in programs are built upon evaluating conditions that return True or False.

Control statements are special instructions in a programming language that are used to alter the normal sequential flow of program execution. In other words, they allow us to decide which statements to execute, when to skip them, or how many times to repeat the execution of a set of statements.

Control statements are broadly classified into:

1. Decision making (selection) statements – Statements that execute selectively based on conditions. It is used to choose between alternative paths.
2. Iteration (looping) statements – Statements that execute a set of instructions repeatedly until a condition is satisfied.

Let us now discuss these statements and their mode of execution.

6.1.1 Decision Making (Selection) Statements

Suppose, we have to find the height of the taller of two students with different heights. Here, two numbers representing the heights of the two students are given as input. To determine who is taller, the two values must be compared so that the appropriate path can be selected to get the correct result. This situation demonstrates the need for decision making in algorithms.

Decision making in algorithms involves choosing one action from two or more alternatives based on a condition. The following algorithm illustrates this concept.

Step 1: Start

Step 2: Input H1, H2 (representing heights of two students)

Step 3: If $H1 > H2$ then

Step 4: Print H1

Step 5: Else

Step 6: Print H2

Step 7: End of If

Step 8: Stop

In Step 3, the condition $H1 > H2$ is evaluated, resulting in either True or False which in turn leads to one of the following.

- If the condition is True, the algorithm executes Step 4 and prints H1 as the taller height. The statements in the Else part are skipped, and the flow moves to Step 8.
- If the condition is False, the algorithm executes Step 6 and prints H2 as the taller height. The statements inside the If block (Step 4) are skipped, and the flow continues to Step 8.

Thus, the algorithm uses a selection mechanism to choose the appropriate action based on the logical result of the condition evaluated in Step 3.

Now, let us depict this algorithm in a pictorial form called flowchart. As we have discussed in Table 3.2 of Chapter 3, a rhombus is used as the decision symbol in flowcharts. It represents a point where a condition is checked and one of two alternative paths is chosen. The general form of an if...else construct in flowcharts is as shown (see Figure 6.1).

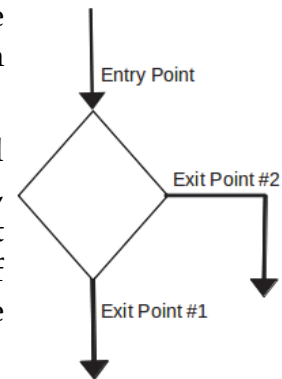


Figure 6.1: if... else construct

The decision symbol has one entry point and two exit points :

- One path for the actions when the condition is True/ Yes.
- Another path for the actions when the condition is False/No.

Now let us try to draw the flowchart of the above algorithm to find the height of the taller student. The flowchart of this algorithm is shown in Figure 6.2. It clearly shows that the program follows only one of the paths, either the “True” path or the “False” path, depending on the condition.

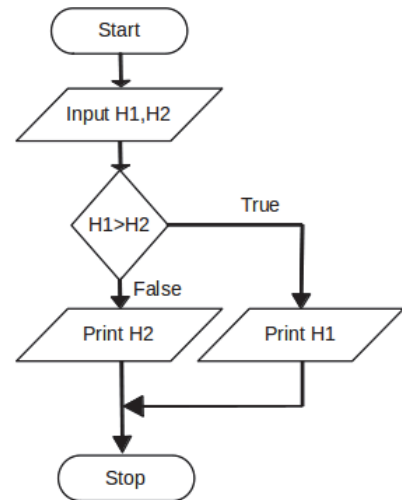


Figure 6.2: Flowchart to print the height of the taller student

6.2 Decision Making (Selection) Statements in Python

In Figure 6.2, we saw how a flowchart can represent decision making visually. In programming, this idea is implemented through control statements, which allow a program to evaluate conditions and direct the flow of execution accordingly. The sequence in which statements run is called the flow of control.

Python supports several decision-making constructs, such as `if`, `elif`, `else`, and the more recent `match-case` structure. The `if-elif-else` statements make choices based on conditions that evaluate to `True` or `False`, whereas the `match-case` construct selects actions by comparing a value against multiple patterns. Together, these tools enable Python programs to choose appropriate actions, skip steps when necessary, and respond flexibly to different inputs, much like how decisions are made in real-life situations.

Types of Decision Making (Selection) Statements in Python

Python provides five types of decision-making statements:

1. `if` statement
2. `if...else` statement
3. `if...elif...else` ladder
4. `match...case` statement
5. Nested `if...else` statement

Let us see each one of them in detail.

6.2.1 if Statement

The `if` statement in Python is used to execute a block of code only when a specified condition is True. If the condition is False, the program skips that block and moves to the next statement. The syntax of `if` statement is

```
if condition:
    block of code
```

A condition in an `if` statement is simply an expression that evaluates to either True or False. Usually it is either a relational or logical expression which we will discuss later in detail. A colon (:) must appear immediately after this condition to indicate the start of the decision block. The statements that belong to the `if` block must be properly indented, as indentation tells Python which lines should be executed when the condition is True. If the condition evaluates to True, the indented statements inside the block are executed; if it evaluates to False, Python skips the entire block and continues with the next statement outside it.

In Python, indentation is mandatory because it defines the scope of a code block under control statements like `if`. Without proper indentation, the program raises an `IndentationError`. This rule ensures that only the indented statements are executed when the condition is True.

Let us understand the use of `if` in problem solving with an example.

Problem 6.1: Suppose the pass score for Computer Science is 30/100, and we need to display a congratulatory message for getting pass score only if the score is greater than or equal to 30. Whatever may be the score, the program should always display the score.

Algorithm:

- Step 1: Start
- Step 2: Input Score
- Step 3: If Score $\geq$ 30 Then
- Step 4: Print "Congratulations! You have passed."
- Step 5: End of If
- Step 6: Print "Your score is:", Score
- Step 7: Stop

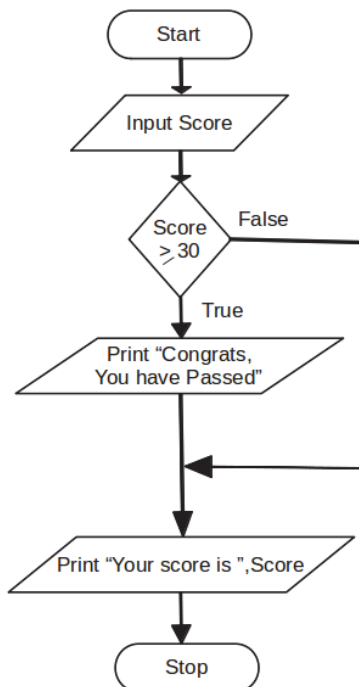


Figure 6.3: Flowchart to display Congratulatory message if eligible

Note that the algorithm and flowchart in Figure 6.3 may differ in minor details. It is up to the programmer to decide what should be included in each. However, the underlying logic to solve the problem must always be preserved. The flowchart section which follows the False part of decision making contains nothing but flow lines as there are no statements to be executed.

A Python program for the above problem is as shown in Figure 6.4. Two different outputs, based on different scores, are provided for clarity. As we can see, the program first checks if the score is greater than or equal to 30; if true, it displays the congratulatory message, otherwise it skips it. Regardless of the result of the condition, the program always displays the score afterwards.

```
pass_score.py - /home/kite/pass_score.py (3.10.12)
File Edit Format Run Options Window Help
score = int(input("Enter the score: "))
if score >= 30:
    print("Congratulations... You scored above pass mark.")
print("Your score is ", score)
Ln: 6 Col: 0
```

```
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
===== RESTART: /home/kite/pass_score.py =====
Enter the score: 28
Your score is 28
>>>
===== RESTART: /home/kite/pass_score.py =====
Enter the score: 35
Congratulations... You scored above pass mark.
Your score is 35
Ln: 12 Col: 0
```

Figure 6.4: Python code to print congratulatory message if qualified

Let us see one more example using `if` in programming.

Problem 6.2: To check whether a student is eligible to attend the exam based on his/her attendance percentage. If the attendance is 75% or above, display "You are eligible to attend the exam".

Algorithm:

- Step 1: Start
- Step 2: Print "Enter your attendance in percentage:"
- Step 3: Input attendance
- Step 4: If attendance ≥ 75 Then
- Step 5: Print "You are eligible to attend the exam."
- Step 6: End of If
- Step 7: Stop

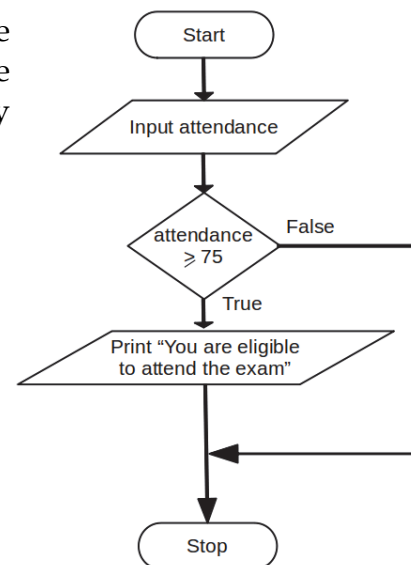


Figure 6.5: Flowchart to check exam eligibility based on attendance percentage.

The program shown in Figure 6.6 uses a simple `if` statement to check a condition. It displays the message "You are eligible to attend the exam." only if the given criteria are met as given in the flowchart (see Figure 6.5).

Consider the program that checks the attendance of students to appear for the exam.

If the student's attendance is less than 75%, the program should inform the student that he/she has to apply for condonation to be eligible for

```
attendance.py - /home/kite/attendance.py (3.10.12)
File Edit Format Run Options Window Help
attendance = int(input("Enter the attendance percentage: "))
if attendance >= 75:
    print("You are eligible to attend the exam")
Ln: 5 Col: 0
```

```
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Enter the attendance percentage: 90
You are eligible to attend the exam
Ln: 8 Col: 0
```

Figure 6.6: Python code to check exam eligibility

appearing for the exams. The `if` statement alone can handle only the True part. When we also want the program to respond to the false condition, we use the second form of decision making statement in Python, the `if...else` construct.

6.2.2 `if ... else` Statement

The `if...else` statement is used when a program needs to make a choice between two alternative actions. The syntax of `if ... else` statement is given below.

```
if condition:
    Statement block 1
else:
    Statement block 2
```

If the condition in the `if` evaluates to True, Statement block 1 is executed; otherwise, Statement block 2 in `else` is executed. This ensures that the program gives an appropriate response for both the possible outcomes of a condition.

Proper indentation is required to clearly indicate the statements that belong to each block. Thus, the `if...else` statement helps in making clear and complete decisions in a program.

Let us modify Problem 6.2 with `if...else` statement

Problem 6.3: Display a message of eligibility when attendance is 75% or above, and a warning message for condonation application when it is below 75%.

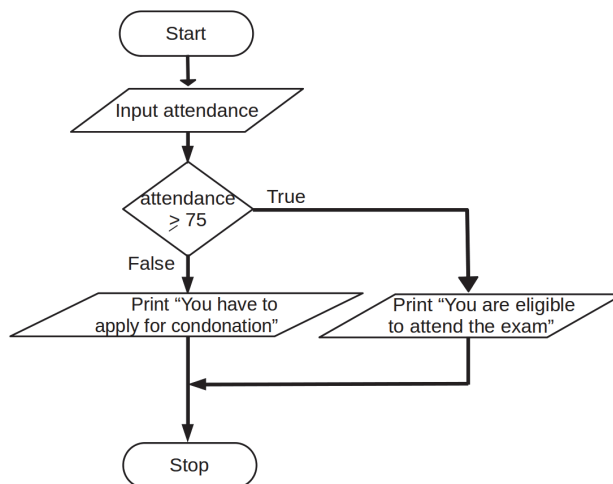


Figure 6.7: Flowchart to print messages based on attendance criteria

Algorithm:

- Step 1: Start
- Step 2: Input attendance
- Step 3: If attendance ≥ 75 Then
- Step 4: Print "You are eligible to attend the exam."
- Step 5: Else
- Step 6: Print "You have to apply for condonation."
- Step 7: End of If
- Step 8: Stop

As shown in Figure 6.7, the condition `attendance >= 75` is evaluated first. If this condition is True, the statement inside the `if` block is executed, displaying the message "You are eligible to attend the exam."

If the condition is False, the program executes the statement inside the `else` block, displaying "You have to apply for condonation."

The program shown in Figure 6.8 uses `if...else` statement to find eligibility for appearing for the exam. As mentioned earlier, proper indentation is used to indicate which statements belong to the `if` block and which belong to the `else` block, ensuring clarity and correctness of the program's logic.

Observe the outputs for different inputs to see the use of `if...else` statement.

The top window shows the code for `attendance_else.py`:

```
attendance_else.py - /home/kite/attendance_else.py (3...
File Edit Format Run Options Window Help
attendance = int(input("Enter your attendance percentage: "))
if attendance >= 75:
    print("You are eligible to attend the exam.")
else:
    print("You have to apply for condonation.")
|
```

The bottom window shows the execution in the IDLE Shell:

```
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
===== RESTART: /home/kite/attendance_else.py =====
Enter your attendance percentage: 97
You are eligible to attend the exam.
>>>
===== RESTART: /home/kite/attendance_else.py =====
Enter your attendance percentage: 68
You have to apply for condonation.
Ln: 11 Col: 0
```

Figure 6.8: Python code to print message based on attendance criteria

Let us revisit section 6.1.1. The algorithm and the corresponding flowchart discussed there (see Figure 6.2) can now be coded in Python using `if...else` statement.

Problem 6.4: Display the greater height for the two inputs.

The Python code in Figure 6.9 is the program statements corresponding to the algorithm and flowchart which we have discussed in Section 6.1.1.

The top window shows the code for `taller.py`:

```
taller.py - /home/kite/taller.py (3.10.12)
File Edit Format Run Options Window Help
h1 = int(input("Enter the height of first student: "))
h2 = int(input("Enter the height of second student: "))
if h1 > h2:
    print("Larger height is: ", h1)
else:
    print("Larger height is: ", h2)
```

The bottom window shows the execution in the IDLE Shell:

```
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
===== RESTART: /home/kite/taller.py =====
Enter the height of first student: 163
Enter the height of second student: 172
Larger height is: 172
>>>
===== RESTART: /home/kite/taller.py =====
Enter the height of first student: 178
Enter the height of second student: 175
Larger height is: 178
Ln: 13 Col: 0
```

Figure 6.9: Python Code to find taller height using `if...else`

Here we can see that only one block in the decision statement is executed, not both, depending on the condition.

In the earlier examples, we used the `if` and `if...else` statements to handle situations where a program needed to choose between two

possible outcomes. However, in many real-life situations, decisions are not limited to just two choices. A program may need to select one option from more than two possibilities based on a condition.

6.2.3 if...elif...else Statement

Often, decisions are not limited to just two choices. In such cases, the program must select one option out of several possible options based on given conditions. The if...elif...else statement is used when there are multiple conditions to test. The conditions are checked from top to bottom. If the first condition is False, the next condition is checked, and so on. As soon as one condition evaluates to True, the corresponding block of statements is executed, and the remaining conditions are skipped. If none of the conditions are True, the else block is executed. The else part is optional and can be omitted when no action is needed for the False conditions. The number of elif statements depends on the number of conditions to be tested. This structure helps in choosing one specific action among several alternatives. The Syntax of if...elif...else statement is

if condition1:

 Statement Block 1

elif condition2:

 Statements Block 2

.....

else:

 Statements Block to execute when all conditions above are False

Problem 6.5: To check whether a given number is positive, negative or zero using the if...elif...else statement.

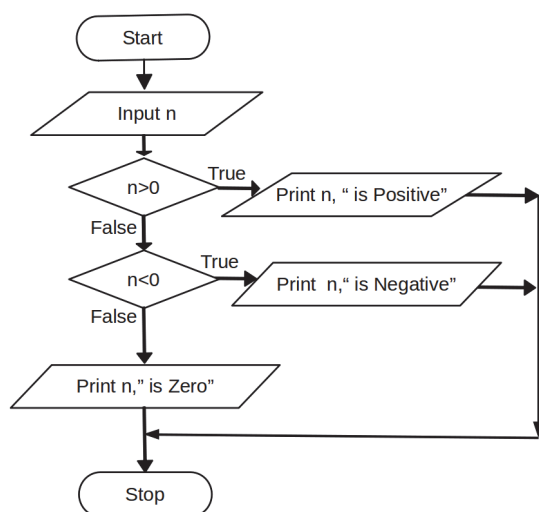


Figure 6.10: Flowchart to check a number Positive, Negative or Zero

Algorithm:

- Step 1: Start
- Step 2: Input n
- Step 3: If n>0 Then
- Step 4: Print n," is Positive"
- Step 5: Else If n<0 Then
- Step 6: Print n," is Negative"
- Step 7: Else
- Step 8: Print n," is Zero"
- Step 9: End of If
- Step 10: Stop

The algorithm above using if - else if - else, the conditions are evaluated in order from top to bottom. Figure 6.10 depicts its flowchart.

Now we are familiar with the if - else if - else construct for multiple branching. Let us see how this construct is implemented in Python.

In the program shown in Figure 6.11, the user is first asked to enter a number, which is stored in the variable `n`. The program then uses a series of conditional statements to determine whether the number is positive, negative or zero. The first condition checks if `n>0`; if this is True, the program prints that the number is positive. If the

```

pos_neg_zero.py - /home/kite/p...
File Edit Format Run Options Window Help
n = int(input("Enter a number: "))
if n > 0:
    print(n, "is positive")
elif n < 0:
    print(n, "is negative")
else:
    print(n, "is zero")
Ln: 9 Col: 0

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
===== RESTART: /home/kite/pos_neg_zero.py =====
Enter a number: 9
9 is positive
>>>
===== RESTART: /home/kite/pos_neg_zero.py =====
Enter a number: -7
-7 is negative
>>>
===== RESTART: /home/kite/pos_neg_zero.py =====
Enter a number: 0
0 is zero
Ln: 15 Col: 0

```

Figure 6.11: Python Code to check a number Positive, Negative or Zero

first condition is False, the program moves to the next condition, `n<0`, and prints that the number is negative if this condition holds True. If neither of the two conditions is True, it means the number must be zero, and the else block prints the corresponding message. Thus, the program clearly identifies the nature of the given number using Python's `if...elif...else` structure.

The `if...elif...else` statement is a powerful tool for selecting one action among many alternatives based on multiple conditions. This structure is particularly useful when decisions depend on ordered comparisons. However, there are situations where decision making does not rely on ordered comparisons but instead depends on matching a value against several fixed patterns or categories. In such cases, Python provides another decision-making construct. Now let us get familiarised with it.

6.2.4 match - case Statement

Python 3.10 and later version introduce an additional decision-making construct called the match-case statement. This structure allows a program to compare a value against several possible patterns and execute the block of code associated with the first matching pattern. It provides a cleaner and more readable alternative when handling multiple distinct choices and functions

somewhat like a switch statement in other languages, but with more powerful pattern-matching capabilities. The match-case construct is particularly useful when the program's decision depends on the value of a single variable, such as selecting a menu option, identifying days of the week, interpreting fixed codes, or determining grades. If none of the specified cases match, the special underscore (_) case can be used as the default action. This improves clarity and simplifies programs that would otherwise require lengthy chains of if...elif statements. The syntax of match-case statement is shown below.

```
match expression:
    case value1:
        Statements Block 1
    case value2:
        Statements Block 2
    case value3:
        Statements Block 3
    .....
    case _:
        Default Statements Block
```

The match keyword checks the given expression, and each case value is compared with it. When a match is found, the corresponding block is executed. In a match-case statement, the values used in value1, value2, etc., can be constants such as integers, strings, tuples, or other literal patterns that are matched against the subject expression.

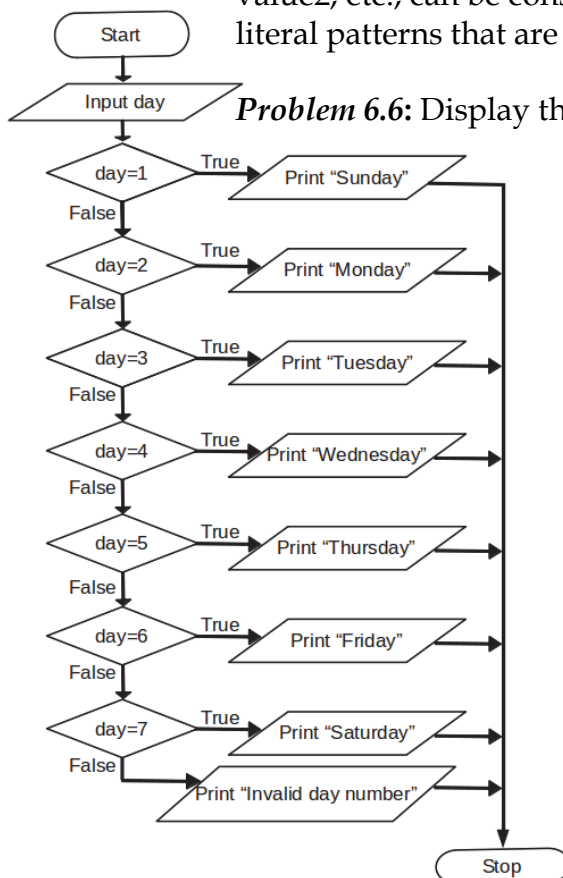


Figure 6.12: Flowchart to print the day of the week

Problem 6.6: Display the day of a week using match - case

Let us consider a simple program that uses the match - case statement to display the name of a day based on a number entered by the user. The user enters a number between 1 and 7, where each number corresponds to a day of the week. For example, 1 represents Sunday, 2 represents Monday, and so on up to 7 representing Saturday. This problem is well suited for the match - case statement because the decision depends on the value of a single variable and involves multiple fixed choices. The algorithm to do this is as given below. Figure 6. 12 shows the flowchart.

Algorithm:

- Step 1: Start
- Step 2: Input day
- Step 3: If day=1 Then
- Step 4: Print "Sunday"

Step 5: Else If day=2 Then
 Step 6: Print "Monday"
 Step 7: Else If day=3 Then
 Step 8: Print "Tuesday"
 Step 9: Else If day=4 Then
 Step 10: Print "Wednesday"
 Step 11: Else If day=5 Then
 Step 12: Print "Thursday"
 Step 13: Else If day=6 Then
 Step 14: Print "Friday"
 Step 15: Else If day=7 Then
 Step 16: Print "Saturday"
 Step 17: Else
 Step 18: Print "Invalid day number"
 Step 19: End of If
 Step 20: Stop

In the program shown in Figure 6.13, the user is first asked to enter a number between 1 and 7, which is stored in the variable day. The match statement then compares the value of day with each case. When a case value matches the input, the corresponding block of statements is executed, and the name of the appropriate day is printed. If the entered number does not match any of the cases from 1 to 7, the underscore (_) case acts as the default option and prints an error message indicating an invalid input. This program demonstrates how the match-case statement simplifies decision making when multiple fixed options are involved, making the code easier to read and maintain compared to long if...elif ladders.

Now let us see another variant of usage of match-case.

The screenshot shows a Python IDE with two windows. The top window, titled 'weedays.py - /home/kite/weedays....', contains the following code:

```

day = int(input("Enter a number (1-7): "))
match day:
    case 1:
        print("Sunday")
    case 2:
        print("Monday")
    case 3:
        print("Tuesday")
    case 4:
        print("Wednesday")
    case 5:
        print("Thursday")
    case 6:
        print("Friday")
    case 7:
        print("Saturday")
    case _:
        print("Invalid day number")
  
```

The bottom window, titled 'IDLE Shell 3.10.12', shows the execution of the program. It displays two runs of the code, each preceded by 'RESTART: /home/kite/weedays.py'. In the first run, the user enters '4' and the output is 'Wednesday'. In the second run, the user enters '8' and the output is 'Invalid day number'.

Figure 6.13: Python code to print the day of the week

Table 6.1: Score range and corresponding grades

Score Range	Grade
90 and above	A+
80-89	A
70-79	B+
60-69	B
50-59	C+
40 - 49	C
30 - 39	D+
20 - 29	D
Below 20	E

Problem 6.7: Read the score in one subject of a student out of 100 and to print the corresponding Grade. Grades are assigned as shown in Table 6.1.

Algorithm:

Step 1: Start
 Step 2: Input score
 Step 3: If score ≥ 90 Then
 Step 4: Print "A+"
 Step 5: Else If score ≥ 80 Then
 Step 6: Print "A"
 Step 7: Else If score ≥ 70 Then
 Step 8: Print "B+"
 Step 9: Else If score ≥ 60 Then
 Step 10: Print "B"
 Step 11: Else If score ≥ 50 Then
 Step 12: Print "C+"
 Step 13: Else If score ≥ 40 Then
 Step 14: Print "C"
 Step 15: Else If score ≥ 30 Then
 Step 16: Print "D+"
 Step 17: Else If score ≥ 20 Then
 Step 18: Print "D"
 Step 19: Else
 Step 20: Print "E"
 Step 21: End of If
 Step 22: Stop

Observe the algorithm that reads the score in one subject of a student out of 100 and to prints the corresponding Grade Flowchart for the same is given in Figure 6.14.

When the program is executed (see Figure 6.15), it first asks the user to enter the score obtained in the subject. The value entered is stored in the variable score, and the match statement compares this value against the specified cases. In the given example, **s** is a pattern variable that takes the value of score in each case of the match statement. It is used with if conditions to check whether the entered score falls within a specific mark range so that the corresponding grade can be printed. If the score is 90 or above, the case corresponding to "A+" is selected, and the program prints "Grade: A+". If the

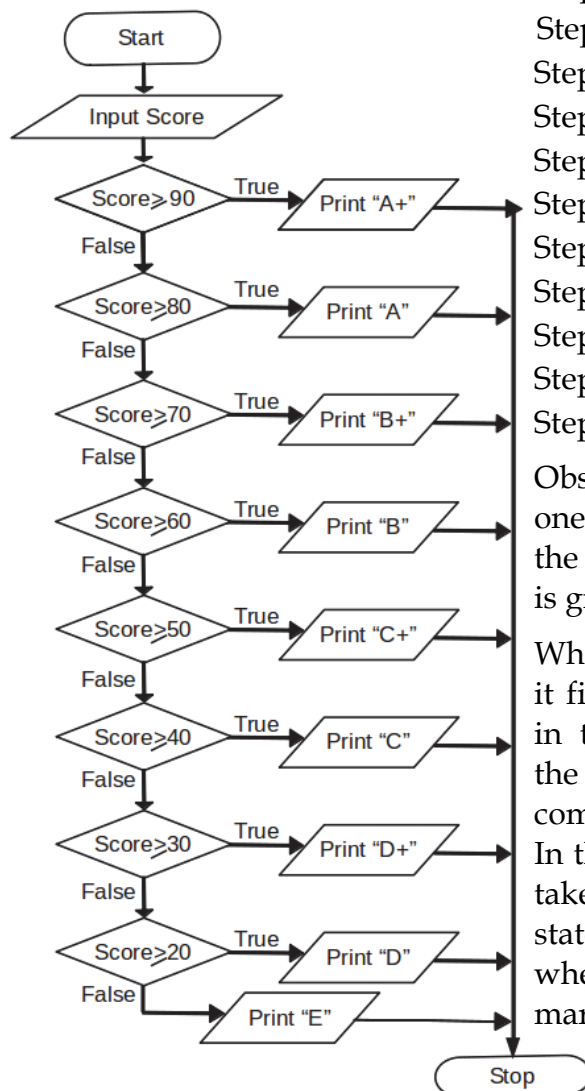


Figure 6.14: flowchart of assigning grades based on scores

score does not meet this condition, Python moves to the next case and checks whether it falls within the 80–89 range, then 70–79, and so on. Each case includes a condition that verifies whether the score lies within a particular grade boundary. As soon as the score satisfies one of these conditions, the matching block is executed and the remaining cases are ignored.

If none of the grade ranges match—for example, if the user enters a value less than 20 or an invalid mark—the default case (__) executes and prints “Invalid score”. Thus, the output displayed by the program depends entirely on the score entered by the user. For example, if the user enters 86, the condition for grade A becomes True, and the program prints “Grade: A”. If the user enters 45, the program identifies the correct range and prints “Grade: C”. This example demonstrates how the match-case statement can be used effectively by grouping multiple values or ranges into appropriate cases, making the program more readable than a long if...elif ladder.

The image shows two windows from the Python IDLE environment. The top window, titled 'grade.py - /home/kite/grade.py (3.10.12)', displays the source code for a program that assigns grades based on a score. The code uses a match-case statement with cases for scores 90-100 (A+), 80-89 (A), 70-79 (B+), 60-69 (B), 50-59 (C+), 40-49 (C), 30-39 (D+), 20-29 (D), 0-20 (E), and a default case for invalid scores. The bottom window, titled 'IDLE Shell 3.10.12', shows the program's execution. It prompts the user to 'Enter the score (out of 100):' and displays the resulting grade for four different inputs: 100 (A+), 54 (C+), 13 (E), and 120 (Invalid Score). Each execution is preceded by a 'RESTART: /home/kite/grade.py' message.

```

grade.py - /home/kite/grade.py (3.10.12)
File Edit Format Run Options Window Help
score = int(input("Enter the score (out of 100): "))
match score:
    case s if 90 <= s <=100:
        print("Grade: A+")
    case s if 80 <= s <=89:
        print("Grade: A")
    case s if 70 <= s <=79:
        print("Grade: B+")
    case s if 60 <= s <=69:
        print("Grade: B")
    case s if 50 <= s <=59:
        print("Grade: C+")
    case s if 40 <= s <=49:
        print("Grade: C")
    case s if 30 <= s <=39:
        print("Grade: D+")
    case s if 20 <= s <=29:
        print("Grade: D")
    case s if 0 <= s < 20:
        print("Grade: E")
    case _:
        print("Invalid Score")
Ln: 24 Col: 0

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
===== RESTART: /home/kite/grade.py =====
Enter the score (out of 100): 100
Grade: A+
>>>
===== RESTART: /home/kite/grade.py =====
Enter the score (out of 100): 54
Grade: C+
>>>
===== RESTART: /home/kite/grade.py =====
Enter the score (out of 100): 13
Grade: E
>>>
===== RESTART: /home/kite/grade.py =====
Enter the score (out of 100): 120
Invalid Score
Ln: 19 Col: 0
  
```

Figure 6.15: Python code using match-case to assign grades

In Python’s match-case construct, the symbol |, known as the vertical bar or pipe, represents an OR (alternative) pattern and is used to specify multiple matching values within a single case clause.

It is part of Python's pattern-matching syntax and allows one case block to match several possible values, making the code more concise and easier to read when different values require the same action. See Figure 6.16 for an example and sample output.

The screenshot shows a Python IDE window titled 'excellent.py - /home/kite/excelle...'. The code in the editor is:

```
score = 10
match score:
    case 10 | 9:
        print("Excellent")
```

The status bar at the bottom of the editor indicates 'Ln: 6 Col: 0'. Below the editor is the 'IDLE Shell 3.10.12' window, which shows the output 'Excellent' and a prompt '>>>>'.

Figure 6. 16: Python code using multiple pattern within a single case

In this code if score is 10 or 9, the same block is executed. The `|` operator can be used only inside a case pattern. Unlike the logical or, it is part of Python's pattern matching syntax, so writing something like `case 10 or 9:` is not valid, because `or` returns a Boolean result, not a pattern. This feature makes `match-case` especially useful for grouping related values together in a clean and readable form.

In certain cases, we need to make a second-level decision only after the first condition is evaluated. In such situations, the decision-making statements discussed earlier (`if`, `if...else`, and `if...elif...else`, `match...case`) may not be sufficient to represent the logic clearly. To handle these cases, Python provides the nested `if...else` statement, in which an `if...else` structure can appear inside the `if` block or the `else` block of another `if...else` statement.

Nesting in Decision Making Statements

Writing one decision-making statement inside any section of another decision-making statement is known as nesting. In other words, repeated use of `if`, `if...else`, `if...elif...else` or `match...case` inside any of these constructs is considered nesting. This means one complete decision-making structure can be placed within another to form a layered decision process. The nested form allows programs to handle more complex situations by making secondary decisions only after a primary condition is satisfied. Thus, nesting helps represent decision paths that depend on the outcomes of previous conditions.

6.2.5 Nested `if...else` Statement

Consider the case of checking whether a given year is a leap year. Determining a leap year requires evaluating multiple related conditions in a particular order. A year is considered a leap year if it is divisible by 4. However, years divisible by 100 are not leap years unless they are also divisible by 400. This means the second and third conditions must be checked only if the previous condition is satisfied. Such dependent decision-making, where one condition leads to another, is best handled using nested `if...else` statements.

A nested `if...else` statement means an `if...else` placed inside another `if` or `else` block. It allows testing multiple related conditions in a hierarchical manner. The placement of `if...else` within an `if` block is shown below.

```

if condition1:
    if condition2:
        Statements when both condition1 and condition2 are True
    else:
        Statements when condition1 is True and condition2 is False
else:
    Statements when condition1 is False

```

In this form, the inner decision is checked only if the outer condition (condition1) is True. The placement of `if...else` within an `else` block is shown below.

```

if condition1:
    Statements when condition1 is True
else:
    if condition2:
        Statements when condition1 is False and condition2 is True
    else:
        Statements when both conditions are False

```

In this form, the second-level decision is made only when the outer condition is False. This structure is useful when different checks are required depending on whether the first condition fails.

Both forms of nested `if...else` help in creating multi-level decision structures. The placement of the inner `if...else` (inside either the `if` or the `else` block) depends on which condition requires further checking. Such nesting makes the program capable of handling complex decision-making situations in an organised and logical manner. Let us see how nesting can be applied to solve leap year checking problem.

Problem 6.8: To check whether a given year is a leap year using a nested `if...else` statement.

Algorithm:

- Step 1: Start
- Step 2: Input year
- Step 3: If the year is divisible by 4 Then
- Step 4: If the year is divisible by 100 Then
- Step 5: If the year is divisible by 400 Then
- Step 6: Print "Leap year"
- Step 7: Else
- Step 8: Print "Not a leap year"
- Step 9: End of If
- Step 10: Else
- Step 11: Print "Leap year"
- Step 12: End of If
- Step 13: Else
- Step 14: Print "Not a leap year"
- Step 15: End of If
- Step 16: Stop

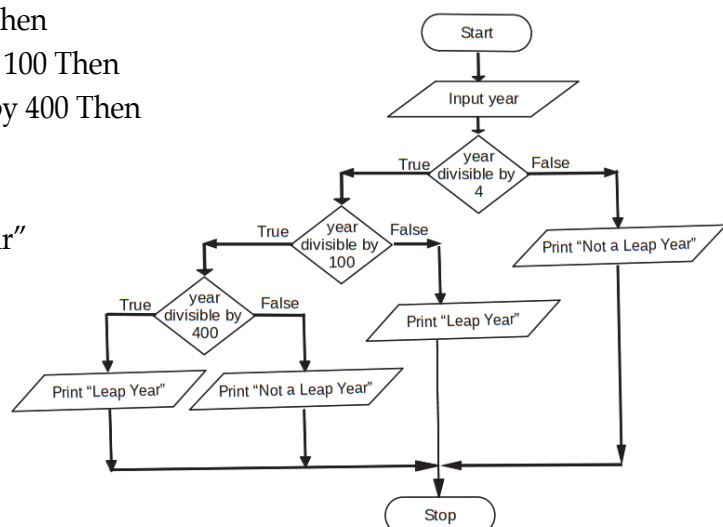


Figure 6.17: Flowchart to check Leap Year or Not

The algorithm and flowchart shown in Figure 6.17 use nested if...else statements to check whether the given year is a leap year or not. The program shown in Figure 6.18 demonstrates how nested if...else statements allow testing of dependent conditions. The outer if checks whether the year is divisible by 4. If this condition is True, a second if checks whether it is divisible by 100. If that is

also True, a third if checks divisibility by 400. If the year is divisible by 400, the program prints "Leap year"; otherwise, it prints "Not a leap year." If the year is divisible by 4 but not by 100, the program directly prints "Leap year." If the year is not divisible by 4, the outer else executes and prints "Not a leap year." This hierarchical decision-making is efficient because each subsequent test depends on the truth of the previous condition, making the program logically structured and easy to follow.

```

leapyear.py - /home/kite/leapyear.py
File Edit Format Run Options Window Help
year = int(input("Enter a year: "))
if year % 4 == 0:
    if year % 100 == 0:
        if year % 400 == 0:
            print("Leap year")
        else:
            print("Not a leap year")
    else:
        print("Leap year")
else:
    print("Not a leap year")
Ln: 13 Col: 0

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
===== RESTART: /home/kite/leapyear.py =====
Enter a year: 2000
Leap year
>>>
===== RESTART: /home/kite/leapyear.py =====
Enter a year: 2001
Not a leap year
>>>
===== RESTART: /home/kite/leapyear.py =====
Enter a year: 2004
Leap year
>>>
===== RESTART: /home/kite/leapyear.py =====
Enter a year: 2100
Not a leap year
Ln: 19 Col: 0

```

Figure 6.18: Python code to check Leap year

Sample test cases of inputs are as shown in Table 6.2, This demonstrates how the nested if...else structure efficiently performs multi-level checking where each condition depends on the previous one, leading to a clear and accurate result.

Table 6.2: Output analysis of the nested if...else leap year program

Input Year	Divisible by 4?	Divisible by 100?	Divisible by 400?	Program Output
2000	Yes	Yes	Yes	Leap year
2001	No	—	—	Not a leap year
2004	Yes	No	—	Leap year
2100	Yes	Yes	No	Not a leap year



Let us do

Develop algorithms, construct flowcharts and write Python code for the following problems.

1. To check whether a given number is even or odd using an `if...else` statement.
2. To accept three numbers and print the largest among them using `if...elif...else`.
3. To check whether a person is eligible for a senior citizen discount ($\text{age} \geq 60$).
4. To accept the total bill amount from the user and apply a 10% discount if the bill is above ₹1000; otherwise, give a 5% discount. Print the final amount.
5. To input a year and check whether it is a century year or not (i.e., divisible by 100).
6. To check whether a student has passed a subject based on the condition:
marks $\geq 30 \rightarrow$ Pass, otherwise Fail.
7. To accept a character from the user and check whether it is a vowel or a consonant using `match...case`.
8. To input the monthly electricity consumption of a household and classify it as:
 - ≤ 100 units $\rightarrow$ Low usage
 - 101–300 units $\rightarrow$ Moderate usage
 - Above 300 units $\rightarrow$ High usage**
9. Each of the following codes contains an error (syntax, indentation, or logic), and the intended purpose of the program is given as a comment. Identify the error and rectify it.

1. Find out the syntax error
 # Purpose: Check whether a number is
 # positive, negative, or zero.
`n = int(input("Enter a number: "))`
`if n > 0`
 `print("Positive")`
`elif n < 0:`
 `print("Negative")`
`else:`
 `print("Zero")`

2. Find out the indentation error
 # Purpose: Print "Eligible" if age is 18 or above,
 # otherwise "Not eligible".
`age = int(input("Enter age: "))`
`if age >= 18:`
 `print("Eligible")`
`else:`
 `print("Not eligible")`

3. Find out the logical error
 # Purpose: Leap year checking.
`year = int(input("Enter year: "))`
`if year % 4 == 0:`
 `print("Leap year")`
`elif year % 100 == 0:`
 `print("Not a leap year")`
`elif year % 400 == 0:`
 `print("Leap year")`
`else:`
 `print("Not a leap year")`

4. Find out the logical error
 # Purpose: Check whether a student has passed
 # (mark greater than or equal to 30).
`mark = int(input("Enter mark: "))`
`if mark = 30:`
 `print("Pass")`
`else:`
 `print("Fail")`

10. Predict the output of the following Python codes.

```
1  n = 7
   if n > 10:
       print("Greater than 10")
   elif n > 5:
       print("Greater than 5")
   else:
       print("5 or below")
```

```
2  score = 85
   match score // 10:
       case 10 | 9:
           print("A+")
       case 8:
           print("A")
       case 7:
           print("B+")
       case _:
           print("Other Grade")
```

```
3  marks = 72
   if marks >= 90:
       print("A+")
   elif marks >= 80:
       print("A")
   elif marks >= 70:
       print("B+")
   else:
       print("C")
```

Selection statements in Python allow a program to choose different execution paths based on conditions. The `if` statement executes a block only when a condition is `True`, whereas the `if...else` statement selects between two alternatives. When multiple conditions must be checked in sequence, the `if...elif...else` structure helps to choose exactly one option from among several. For situations where one decision depends on the outcome of another, nested `if...else` statements are used. Python 3.10 and above also provide the more powerful `match-case` construct, which simplifies decisions based on fixed values or patterns and improves readability. Together, these decision-making tools help programs respond intelligently to different inputs and situations. While selection statements allow a program to make one-time choices, many real-world problems require repeating a set of actions multiple times until a condition changes. For such tasks, Python provides iterative or looping statements, which we explore in the upcoming sections.

6.3 Iteration / Looping

Imagine a student calculating the total and average marks of six subjects. The student adds each subject's score one by one and, after completing all six, divides the total by 6 to find the average. This repeated action of adding scores until all subjects are processed illustrates the idea of iteration.

In programming, iteration (or looping) refers to executing a set of statements repeatedly as long as a specified condition remains true. A loop allows a computer to automate repetitive tasks without rewriting instructions. The repetition ends automatically when the condition becomes false. This makes programs efficient, organised, and easy to maintain. While selection statements make one-time decisions, iteration enables repeated execution based on conditions.

Problem 6.9: Printing numbers from 1 to 10.

As shown in the following algorithm, the loop starts with $i = 1$ (initialisation). As long as $i \leq 10$ (condition), the number is printed (body), and then i is incremented by 1 (updation). When i exceeds 10, the loop ends automatically.

Algorithm:

- Step 1: Start
 Step 2: $i = 1$
 Step 3: Repeat Steps 4 and 5 While $i \leq 10$
 Step 4: Print i
 Step 5: $i = i + 1$
 Step 6: Stop

Flowchart represent loops using rhombus and flow lines that are connected in a cycle as shown in Figure 6.19. If the condition is True, the flow moves to the loop body; after execution, it returns to the decision point for re-evaluation.

If the condition becomes False, the loop is exited. The general form shown in Figure 6.20 shows how repetition works within a process.

In the flowchart shown in Figure 6.19, the decision box checks whether $i \leq 10$. If True, the flow moves to the box that prints the value of i , then proceeds to the updation step ($i = i + 1$), and finally loops back to the decision box. If i becomes greater than 10, the condition fails and the flow exits the loop, ending the process. Thus, the flowchart provides a clear visual representation of iterative execution, to understand the concept before translating it into actual code. Loops become even more useful when we need to perform repeated calculations on multiple values.

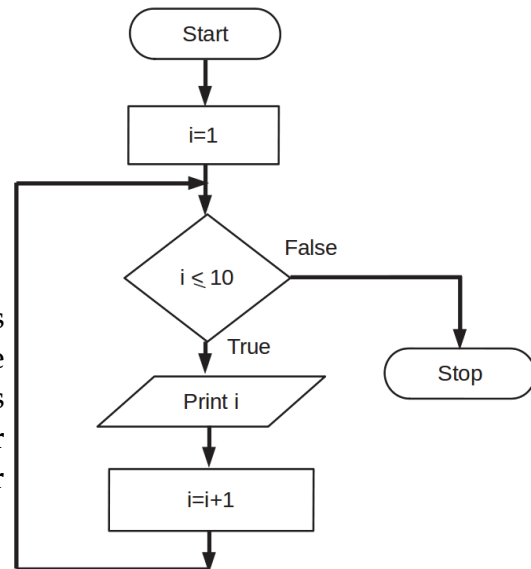


Figure 6.19: Flowchart to print numbers from 1 to 10

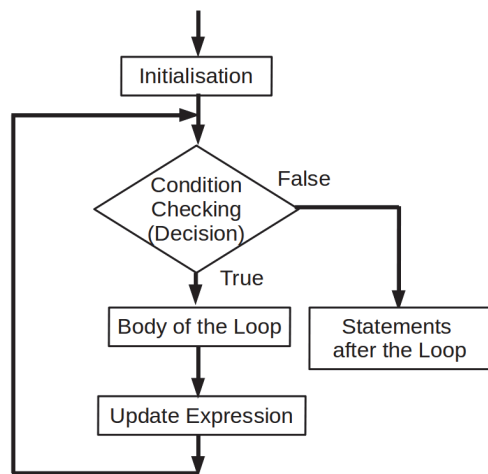


Figure 6.20: General form of a flowchart with loop

Problem 6.10: Finding the sum and average of scores of 6 subjects

Let us consider the problem of finding the sum and average of the scores obtained in 6 subjects. Instead of writing separate steps for each subject, we can use a loop to repeatedly input each score and accumulate the total.

Algorithm:

- Step 1: Start
 Step 2: Set $i = 1$, $\text{sum} = 0$
 Step 3: Repeat Steps 4 to 6 While $i \leq 6$
 Step 4: Input score
 Step 5: $\text{sum} = \text{sum} + \text{score}$
 Step 6: $i = i + 1$
 Step 7: Calculate average = $\text{sum} / 6$
 Step 8: Print sum and average
 Step 9: Stop

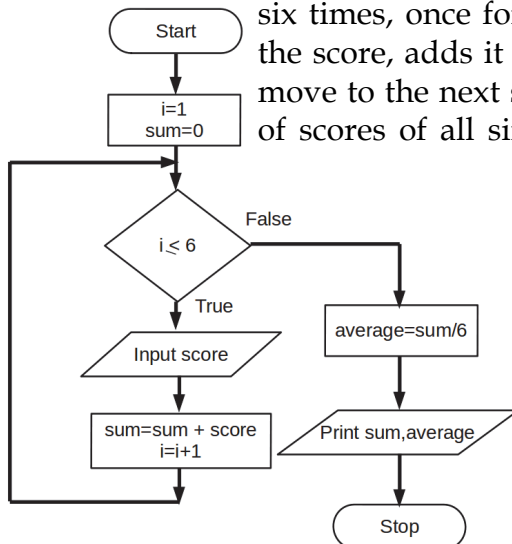


Figure 6.21: Flowchart to find sum and average of 6 subjects

In the algorithm, the loop begins with the initialisation of $i = 1$ and $\text{sum} = 0$. The condition $i \leq 6$ ensures that the loop runs exactly six times, once for each subject. During each cycle, the loop reads the score, adds it to the running total (sum), and then updates i to move to the next subject. When the loop finishes after finding sum of scores of all six subjects, the average is calculated by dividing the value of sum by 6. This example shows how loops help avoid repetition in algorithms by processing multiple values systematically.

The flowchart shown in Figure 6.21, uses a decision box to control the repetition. This flowchart demonstrates how loops handle repeated tasks such as reading multiple inputs and calculating cumulative values in a structured manner. Thus, algorithms and flowcharts use specific symbols and keywords to show how loops repeat a set of instructions.

Understanding how loops appear in these representations helps us visualise the control flow more clearly before converting the logic into actual Python code. Now that we have become familiar with the use of loops in algorithms and flowcharts, we will not include them explicitly for every program discussed hereafter. You can practise drawing the corresponding algorithms and flowcharts on your own to strengthen your understanding.

6.3.1 Parts of a Loop

The steps in algorithm of Problem 6.10 can be broken down into key components that also define a loop in programming:

- **Initialisation:** The variable total is set to zero, and a counter (e.g., $i = 1$) is started.
- **Condition checking:** Before each cycle, it is checked whether any subjects remain (e.g., $i \leq 6$).
- **Body of the loop:** The current subject's mark is added to the total.
- **Updation:** The counter is increased (e.g., $i = i + 1$) to move to the next subject.

This cycle continues until all subjects are processed. When the condition becomes False, the loop stops and the average is calculated.

A similar pattern appears when a teacher identifies the highest total score in a class. Beginning with roll number 1 (initialisation), the teacher checks whether more students remain (condition), compares totals and updates the highest (body), then moves to the next roll number (updation). The process repeats until all students are evaluated.

In programming, every loop follows these same four components, which together enable systematic and controlled repetition of tasks.

6.4 Loops in Python

Just as algorithms and flowcharts use repetition to perform tasks multiple times, Python also provides special statements to repeat a block of code. These are known as **looping statements**. In Python, loops make it possible to execute a group of statements repeatedly without writing the same code again and again. This is especially useful when dealing with tasks such as processing multiple inputs, generating number sequences, performing cumulative calculations, or repeating an action as long as a condition is met. Python supports two main types of loops: **while** loop and **for** loop.

Both **while** and **for** loops repeatedly execute a block of statements and rely on a condition to decide when the repetition should stop. Their behaviour closely follows the logical structure used in algorithms and flowcharts: initialisation to set the starting point, condition checking to decide whether to continue, a loop body containing the repeated statements, and an updation step that moves the loop toward termination. However, in Python, these four parts may not always appear explicitly or in the same place, initialisation and updation often occur outside or inside the loop in different forms, yet conceptually they exist in every loop.

6.4.1 while Loop

The **while** loop is also known as an indefinite loop because it is used when the number of repetitions is not known in advance and depends entirely on a condition. The **while** loop continues to execute as long as the condition remains **True** and stops only when the condition becomes **False**. This makes the **while** loop suitable for situations where the exact number of iterations cannot be predicted beforehand.

Its general form is:

```
while condition:
    statements
```

In the general form, condition is an expression that is evaluated before every iteration. If the condition is **True**, the statements inside the loop are executed. After executing the loop body, the condition is checked again. When the condition becomes **False**, the loop stops and the control moves to the next statement after the loop.

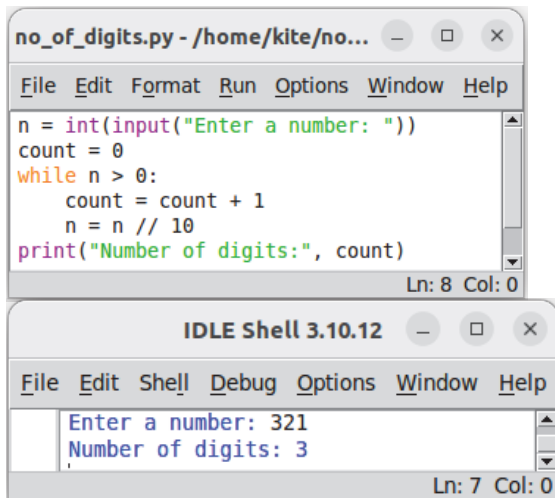


You may notice that initialization (starting value) and updation (changing the loop variable) are not part of this general syntax. This is because Python allows complete flexibility—initialisation and updation are not built into the loop structure but are written separately by the programmer, depending on the logic of the program.

Let us get familiar with **while** loop, with examples.

Problem 6.11: Finding the number of digits in a given number.

Since the **while** loop is an indefinite loop, it is best illustrated using problems where the number of repetitions is not known in advance.



The screenshot shows two windows from an IDE. The top window, titled 'no_of_digits.py', contains the following Python code:

```
n = int(input("Enter a number: "))
count = 0
while n > 0:
    count = count + 1
    n = n // 10
print("Number of digits:", count)
```

The bottom window, titled 'IDLE Shell 3.10.12', shows the execution of the code. It displays the prompt 'Enter a number: 321' followed by the output 'Number of digits: 3'.

Figure 6.22: Python code to count number of digits

A suitable example is counting the number of digits in a given number, as the number of digits depends on the value entered by the user and cannot be predicted beforehand. In this approach, the program repeatedly removes the last digit of the number and increments a counter. This process continues until the number becomes zero.

In the program shown in Figure 6.22, the user enters a number that is stored in the variable **n**, and the variable **count** is initialised to zero. The while loop checks whether **n** is greater than zero. As long as this condition

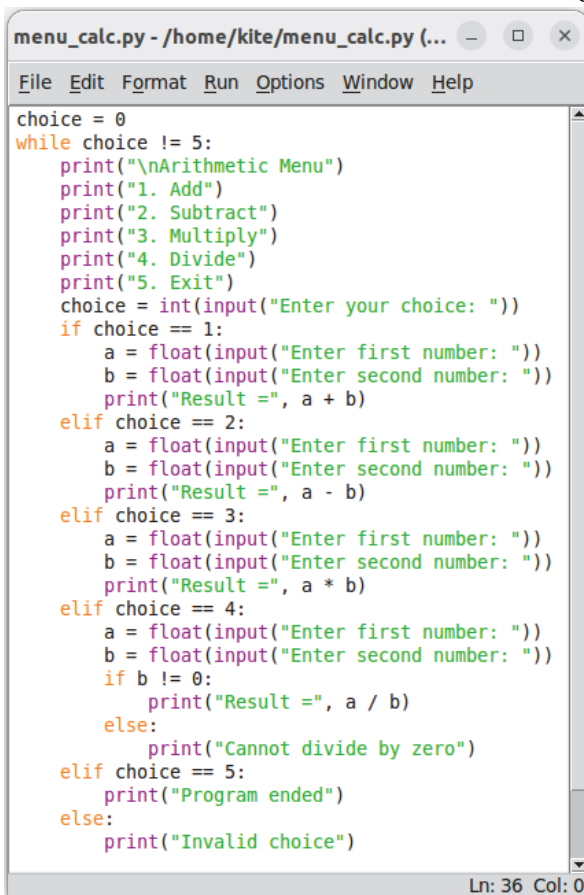
is True, the loop executes, increments the value of **count**, and updates **n** by removing its last digit using integer division ($n//10$). Each iteration of the loop removes one digit from the number. When all digits are removed and **n** becomes zero, the condition becomes False and the loop terminates. The final value of **count** represents the total number of digits in the given number.

For example, if the input is 321, the loop executes three times: first reducing the number to 32, then to 3, and finally to 0. Accordingly, the value of **count** becomes 3, which correctly indicates that the number 321 has three digits. This example clearly demonstrates why

the while loop is suitable for problems where the number of repetitions depends on the input value.

Problem 6.12: Menu-driven calculator using while loop to perform the four basic arithmetic operations - addition, subtraction, multiplication, and division - based on the user's choice. The program should repeatedly display a menu and continue performing operations until the user chooses to exit.

This program (see Figure 6.23(a)) uses a while loop to repeatedly display a menu of arithmetic operations. The loop continues to execute as long as the value of **choice** is not equal to 5. During each iteration, the user is asked to select an operation. If the user selects addition, subtraction, multiplication, or division, the program accepts two numbers and performs the corresponding calculation. For division, an additional check is included to prevent division by zero.



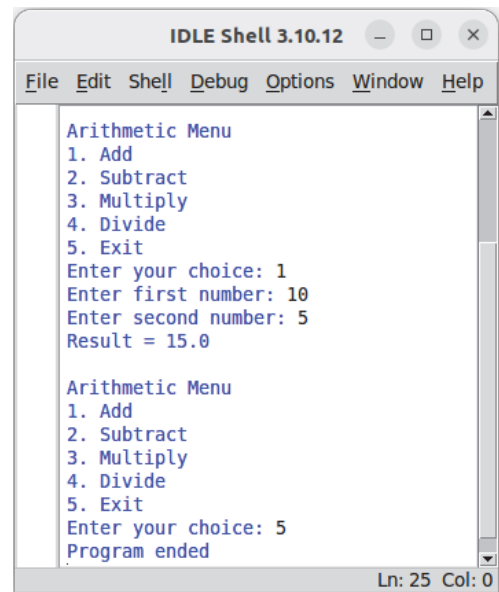
The screenshot shows a window titled 'menu_calc.py' with the following Python code:

```
choice = 0
while choice != 5:
    print("\nArithmetic Menu")
    print("1. Add")
    print("2. Subtract")
    print("3. Multiply")
    print("4. Divide")
    print("5. Exit")
    choice = int(input("Enter your choice: "))
    if choice == 1:
        a = float(input("Enter first number: "))
        b = float(input("Enter second number: "))
        print("Result =", a + b)
    elif choice == 2:
        a = float(input("Enter first number: "))
        b = float(input("Enter second number: "))
        print("Result =", a - b)
    elif choice == 3:
        a = float(input("Enter first number: "))
        b = float(input("Enter second number: "))
        print("Result =", a * b)
    elif choice == 4:
        a = float(input("Enter first number: "))
        b = float(input("Enter second number: "))
        if b != 0:
            print("Result =", a / b)
        else:
            print("Cannot divide by zero")
    elif choice == 5:
        print("Program ended")
    else:
        print("Invalid choice")
```

Figure 6.23(a): Menu driven simple calculator

For example (see Figure 6.23(b)), if the user enters the choice 1 (Addition) and then enters the numbers 10 and 5, the program calculates and displays the result 15.0. After displaying the result, the menu is shown again, allowing the user to perform another operation. If the user enters the choice 5, the message "Program ended" is displayed and the loop condition becomes false. As a result, the loop terminates and the program stops execution.

Since the number of times the menu is displayed depends entirely on the user's choices and cannot be predicted in advance, this program clearly demonstrates the use of the while loop as an indefinite loop.



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Arithmetic Menu
1. Add
2. Subtract
3. Multiply
4. Divide
5. Exit
Enter your choice: 1
Enter first number: 10
Enter second number: 5
Result = 15.0

Arithmetic Menu
1. Add
2. Subtract
3. Multiply
4. Divide
5. Exit
Enter your choice: 5
Program ended
Ln: 25 Col: 0

```

Figure 6.23(b): Output of program shown in Figure 6.23(a)



Let us do

Develop algorithms, construct flowcharts and write Python code for the following problems using while loop.

1. To print the numbers from 1 to 20.
2. To display all even numbers between 1 and 50.
3. To find the sum of the first 10 natural numbers.
4. To print the multiplication table of a given number (for example, 5×1 to 5×10).
5. To accept a number and to check whether it is a perfect number or not.
6. To reverse the digits of a given number (e.g., input 123 $\rightarrow$ output 321).
7. To find the factorial of a number entered by the user.
8. To find the sum of digits of a given number.
9. To check whether a given number is a palindrome.
10. Keep accepting numbers from the user and find their total until the user enters 0. (Hint: use 0 as the loop termination condition.).
11. Predict the output of the following Python codes.

```

1. i = 1
   while i <= 5:
       print(i)
       i = i + 1

```

```

2. i = 10
   while i >= 7:
       print("Hi")
       i = i - 1

```

```

3. i = 2
   while i <= 10:
       print(i, end=" ")
       i = i + 3

```

```

4. i = 1
   total = 0
   while i <= 5:
       total = total + i
       i = i + 1
   print(total)

```

```

5. i = 1
   while i < 20:
       if i % 4 == 0:
           print(i, end=" ")
       i = i + 3

```

```

6. i = 5
   while i > 0:
       print(i)
       i = i - 2
   print("End")

```

12. Each of the following codes contains an error (syntax, indentation, or logic), and the intended purpose of the program is given as a comment. Identify the error and rectify it.

```
1  # Purpose: Print numbers
   # from 1 to 5
   i = 1
   while i <= 5
       print(i)
       i = i + 1
```

```
2  # Purpose: Print even numbers
   # from 2 to 20
   i = 2
   while i <= 20:
       print(i)
       i = i + 1
```

```
4  # Purpose: Reverse a given number
   n = int(input("Enter a number: "))
   rev = 0
   while n > 0:
       rev = rev * 10 + n // 10
       n = n // 10
   print("Reversed number =", rev)
```

```
5  # Purpose: Calculate factorial of
   # a number entered by the user
   n = int(input("Enter a number: "))
   fact = 0
   while n > 0:
       fact = fact * n
       n = n - 1
   print("Factorial =", fact)
```

6.4.2 for Loop

The for loop is a definite loop, used when the number of repetitions is known beforehand, for example, when iterating through a sequence or a fixed range of values. The for statement is particularly useful for iterating through a range of values or the elements of data structures such as strings, lists, or tuples. Since these structures contain a finite number of items, the for loop allows each element to be processed one by one in a clear and predictable manner. These data structures will be explained in the coming chapters, where the for loop will play an important role in accessing and handling their contents efficiently.

Types of for Loops

The for loop in Python can be used in different situations depending on the type of sequence being processed or the nature of the task. The commonly used forms of the for loop are:

a. for loop with a numeric range

In this form, the for loop uses the range() function to repeat a block of code a specific number of times. It is widely used in Python for generating number sequences, performing calculations a fixed number of times, or iterating through index values.

• range() with one parameter

The simplest form of the range() function is: range(n)

Syntax of this type of for loop: for variable in range(n):
statement(s)

This form generates a sequence of n numbers, starting from 0 and going up to $n-1$. It does not include the number n itself because the stop value is always excluded.

Problem 6.13: Print numbers from 0 to 4 using `range()` function with one parameter.

Python code to do this is shown in Figure 6.24. Although the value 5 is given in `range(5)`, the sequence produced ends at 4 because `range(n)` generates numbers from 0 upto $n - 1$.

This form is especially useful when a program requires a counter starting from zero, needs to repeat a task a fixed number of times, or must access elements using index values in lists or strings which we will see in the coming chapters.

• `range()` with two parameters

The next form of the `range()` function is: `range(start, stop)`

Syntax of this type of for loop:

```
for variable in range(start, stop):
    statement(s)
```

When the `range()` function is used with two parameters, `range(start, stop)`, it generates a sequence of numbers beginning from the value specified by `start` and continues up to, but not including, the value specified by `stop`. The start value indicates where the sequence begins, and the stop value determines the boundary at which the sequence ends.

The python code for Problem 6.9 discussed before can also be done using this form of for loop as shown in Figure 6.25.

The output obtained can also be seen in the figure. `range (1,11)` generates the numbers from 1 to 10, representing the roll numbers of the students. During each iteration, the variable `roll` takes the next value in the sequence and prints it. This makes the program shorter, clearer, and easier to adjust if the class size changes.

• `range()` with three parameters

The final form of the `range()` function is: `range(start, stop, step)`

Syntax of this type of for loop:

```
for variable in range(start, stop, step):
    statement(s)
```

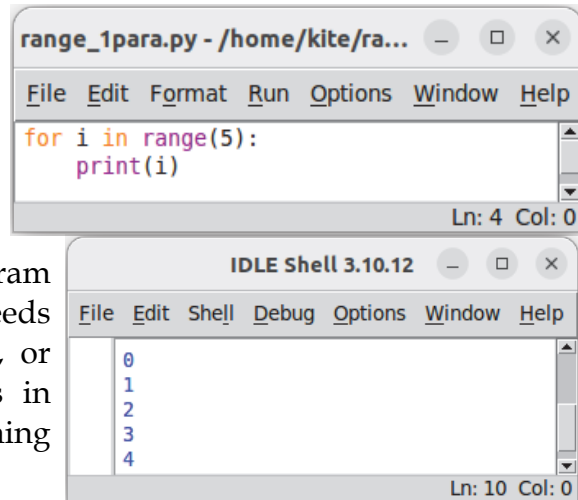


Figure 6.24: Print using `range(n)`

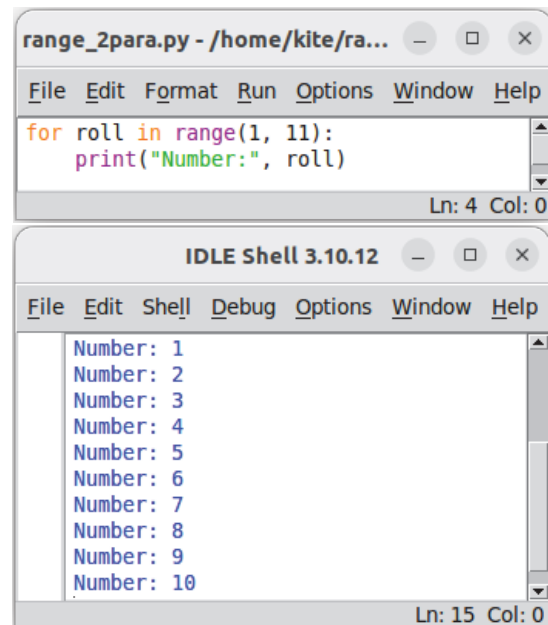
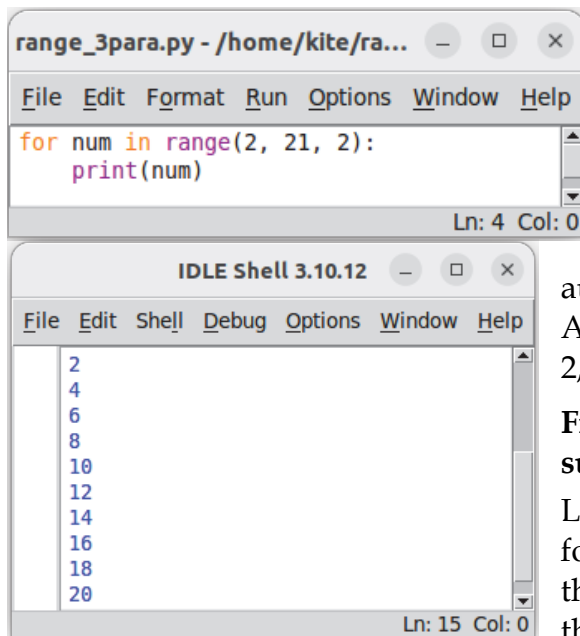


Fig 6.25: Print using `range(start, stop)`

The `range()` function can take an optional third argument called the step value, which determines how much the loop variable changes after each iteration. When no step is given, the loop progresses by 1 each time. By specifying a different step, the loop can move in larger intervals, such as 2 for generating even numbers or even in reverse using a negative value. If the start value is greater than the stop value and the step is positive, the `range()` function produces an empty sequence and the loop does not execute. To iterate in reverse order when `start > stop`, a negative step value must be specified. This makes the for loop highly flexible, enabling it to produce various numeric patterns like even numbers, odd numbers, multiples, or countdowns without additional conditional logic.



```

range_3para.py - /home/kite/ra...
File Edit Format Run Options Window Help
for num in range(2, 21, 2):
    print(num)
Ln: 4 Col: 0

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
2
4
6
8
10
12
14
16
18
20
Ln: 15 Col: 0

```

Figure 6.26: Print using range (start, stop, step)

Problem 6.14: Printing Even Numbers from 2 to 20.

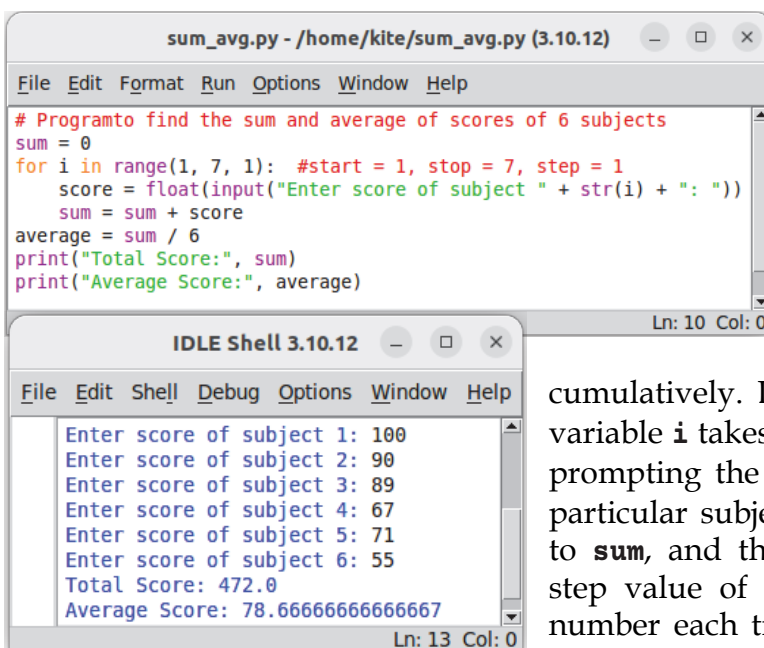
In the program shown (see Figure 6.26), the loop begins at 2 and runs up to 20. A step value of 2 ensures that the loop variable jumps by 2 on each iteration, automatically selecting only even numbers. As a result, the output becomes the sequence 2, 4, 6, ..., 20.

Finding the sum and average of scores of 6 subjects

Let us revisit Problem 6.10. The python code for this example is shown in Figure. 6.27. In this program, the variables used to control the loop and to perform the calculation are

initialised before the loop begins. The loop control variable `i` is initialised implicitly through the `range(1, 7, 1)` function, which generates the sequence 1 to 6, while the program logic variable `sum` is explicitly initialised to 0 so that the scores can be added

cumulatively. During each iteration, the loop variable `i` takes the next value in the sequence, prompting the user to enter the score of that particular subject. The entered score is added to `sum`, and the loop progresses because the step value of 1 moves `i` to the next subject number each time. After all six subjects have been processed, the program calculates the



```

sum_avg.py - /home/kite/sum_avg.py (3.10.12)
File Edit Format Run Options Window Help
# Program to find the sum and average of scores of 6 subjects
sum = 0
for i in range(1, 7, 1): #start = 1, stop = 7, step = 1
    score = float(input("Enter score of subject " + str(i) + ": "))
    sum = sum + score
average = sum / 6
print("Total Score:", sum)
print("Average Score:", average)
Ln: 10 Col: 0

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Enter score of subject 1: 100
Enter score of subject 2: 90
Enter score of subject 3: 89
Enter score of subject 4: 67
Enter score of subject 5: 71
Enter score of subject 6: 55
Total Score: 472.0
Average Score: 78.66666666666667
Ln: 13 Col: 0

```

Figure 6.27: Printing sum and average of 6 subjects

average by dividing the **sum** by 6 and then displays both the **sum** and the **average**. This illustrates how initialising the loop control variable and calculation variables before entering the loop ensures correct and systematic processing within the loop.

b. for loop with strings

A string is a sequence of characters, and the for loop can access each character one by one. This is useful for tasks such as counting characters, checking for specific letters, or processing text character by character.

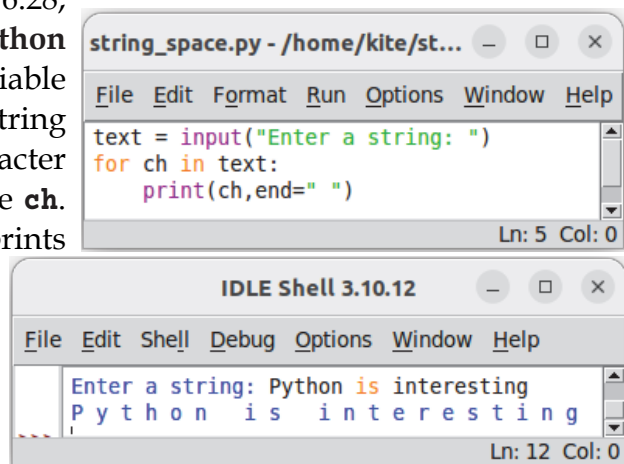
Syntax of using for in strings is:

```
for variable in string:
    statements
```

Variable is the loop variable that takes one character from the string during each iteration. The statements inside the loop are executed once for every character in the string.

Problem 6:15 Print the given string with a blank space inserted after each character.

In the program shown in Figure 6.28, when the user enters the string "Python is interesting", it is stored in the variable `text`. The for loop then processes this string one character at a time, with each character temporarily assigned to the loop variable `ch`. During every iteration, the program prints the current character followed by a blank space using `print(ch, end=" ")`. This ensures that instead of moving to a new line after each character, the output continues on the same line with a space inserted after every character (see Figure 6.28). We will learn more about strings in Chapter 8.



The screenshot shows two windows from an IDE. The top window, titled 'string_space.py - /home/kite/st...', contains the following code:

```
text = input("Enter a string: ")
for ch in text:
    print(ch, end=" ")
```

The bottom window, titled 'IDLE Shell 3.10.12', shows the execution of the program. It displays the prompt 'Enter a string: ' followed by the user input 'Python is interesting'. The output of the program is 'P y t h o n i s i n t e r e s t i n g', where each character is printed followed by a space.

Figure 6.28: Printing string with spaces in between characters

c. for loop with lists, tuples and other collections

Lists, tuples, and similar collections contain multiple values arranged in order. The for loop can process each item in these collections sequentially. This form of the loop is commonly used for handling related groups of data, such as marks of students, items in a shopping list, or values stored for further calculations. These data structures and their operations will be explained in detail in later chapters, where the for loop will play an important role in accessing and manipulating their elements. We will learn the use of for loop with these collections in the coming chapters.



Let us do

Develop algorithms, construct flowcharts and write Python code for the following problems using for loop.

1. To display the first 15 natural numbers.
2. To print the squares of numbers from 1 to 10.
3. Display the sum of even numbers from 2 to 20 using a for loop and the range() function.
4. To print the cube of each number from 1 to 10 using a for loop.
5. To count how many numbers between 1 and 50 are divisible by 3 or 5 using a for loop.
6. To find the product of all numbers in a list entered by the user.
7. To display all numbers between 100 and 200 that are multiples of 7 using a for loop.
8. To find the factorial of a number using a for loop.
9. To find the largest number in a list entered by the user using a for loop.
10. To count how many numbers in a given list are greater than a given number.
11. Each of the following codes contains an error (syntax, indentation, or logic), and the intended purpose of the program is given as a comment. Identify the error and rectify it.

```
1 # Purpose: To print numbers
  # from 1 to 5
  for i in range(1, 5)
    print(i)
```

```
2 # Purpose: To print even numbers
  # from 2 to 10
  for i in range(2, 11, 2):
    print(i)
```

```
3 # Purpose: To print numbers
  # from 1 to 10
  for i in range(1, 11):
    print(i)
    i = i + 1
```

```
4 # Purpose: To print numbers
  # from 10 to 1
  for i in range(10, 1):
    print(i)
```

12. Predict the output of the following Python codes.

```
1 for i in range(5):
    print(i)
```

```
2 for i in range(1, 6):
    print(i, end=" ")
```

```
3 for i in range(10, 5, -2):
    print(i)
```

Table 6.3 shows comparison between for and while loops in Python.

Table 6.3: Comparison between for and while

Aspect	for Loop	while Loop
Nature of Loop	Definite loop - used when the number of repetitions is known.	Indefinite loop - used when repetitions depend on a condition.
Control Mechanism	Iterates over a sequence (range, list, tuple, string).	Repeats execution as long as the condition remains true.
Initialisation & Updation	Usually handled automatically (e.g., by range()).	Must be written manually before and inside the loop.

Condition Checking	Implicitly controlled by the sequence or range.	Explicitly tested before every iteration.
Typical Use Cases	Suitable for fixed counts, sequences, tables.	Suitable for input-driven or condition-driven loops.



A special situation that programmers must avoid is the infinite loop, which occurs when the loop's condition never becomes false. This often happens when the updation step is missing or incorrect, preventing the loop from making progress toward termination. Infinite loops cause programs to continue running without stopping, potentially freezing or slowing down the system. It is therefore important to design loops carefully, ensure correct updation inside the loop body, and confirm that the loop's condition will eventually become false. Understanding these aspects helps prevent unintended infinite loops and ensures that programs run efficiently and safely.

In Python, the underscore `_` is commonly used in a `for` loop when the loop variable is not needed inside the loop body. It acts as a placeholder to show that the value generated in each iteration is intentionally ignored. This is useful when we simply want to repeat an action a certain number of times without using the loop variable. For example, consider the following code.

```
for _ in range(5):
    print("Hello")
```

In this program, "Hello" is printed five times, and the underscore indicates that the loop variable is not required for any computation. This makes the code cleaner and more readable.

The `for` and `while` loops in Python are used to repeatedly execute a block of code based on a condition. The `for` loop is known as a definite loop because it is used when the number of repetitions is known in advance, such as iterating through a range of values or the elements of a sequence like a list or string. In contrast, the `while` loop is an indefinite loop, used when the number of repetitions is not known beforehand and depends on a condition being true. Both loops follow the same logical structure: initialisation of variables, condition checking, execution of the loop body, and updation to ensure termination. Although these parts may not always be explicitly visible in Python code, they are conceptually present in every loop. Together, `for` and `while` loops allow programs to automate repetitive tasks efficiently, making code more organised, flexible, and easier to maintain.

6.5 Loop Control Statements

Loop control statements are special commands in Python that modify the normal flow of a loop. Under usual circumstances, a loop runs repeatedly until its condition becomes False. However, there are situations where we may want to stop a loop early, skip certain iterations, or define what should happen when the loop finishes naturally. Python provides four loop control statements for this purpose.

The four important keywords used with loops in Python are:

- `break` is used to exit a loop immediately, even if the loop's condition has not become False.

- `continue` skips the remaining statements in the current iteration and moves to the next iteration of the loop.
- `pass` acts as a placeholder that does nothing; it is used when a statement is syntactically required but no action is needed.
- `else with loops` executes a block of code only when the loop completes normally without encountering a `break`.

Let us discuss each of them in detail.

6.5.1 `break` Statement

The `break` statement is used to terminate a loop before its natural end. When Python encounters a `break` inside a loop, it immediately stops further iterations and transfers control to the first statement that follows the loop.

```

loop_break.py - /home/kite/loop_b...
File Edit Format Run Options Window Help
for roll in range(1, 11):
    print("Roll Number:", roll)
    if roll == 5:
        print("Stopping attendance check.")
        break
Ln: 7 Col: 0

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Roll Number: 1
Roll Number: 2
Roll Number: 3
Roll Number: 4
Roll Number: 5
Stopping attendance check.
Ln: 20 Col: 0

```

Figure 6.29: Using `break` in for loop

Problem 6.16: Stop the loop when roll number 5 is reached using `break` in a for Loop.

In the program shown in Figure 6.29, the for loop prints roll numbers from 1 to 5.

When the value of `roll` becomes 5, the condition `roll==5` is True, and the `break` statement immediately stops the loop. The remaining roll numbers are not processed, and the program exits the loop (see Figure 6.29).

Problem 6.17: Stop the loop when roll number 5 is reached (Using `break` inside a while loop).

In Python program given in Figure 6.30(a), even though a while loop to print roll numbers from 1 to 10 is used, it stops the loop when roll number 5 is reached. It uses the `break` statement inside the loop to terminate it prematurely.

In this program, the variable `roll` is initialised to 1, marking the starting point of the loop. The while loop continues executing as long as the condition `roll<=10` remains True.

```

while_break.py - /home/kite/while...
File Edit Format Run Options Window Help
roll = 1
while roll <= 10:
    print("Roll Number:", roll)
    if roll == 5:
        print("Stopping attendance check.")
        break
    roll = roll + 1
Ln: 9 Col: 0

```

Figure 6.30(a): Using `break` in while loop

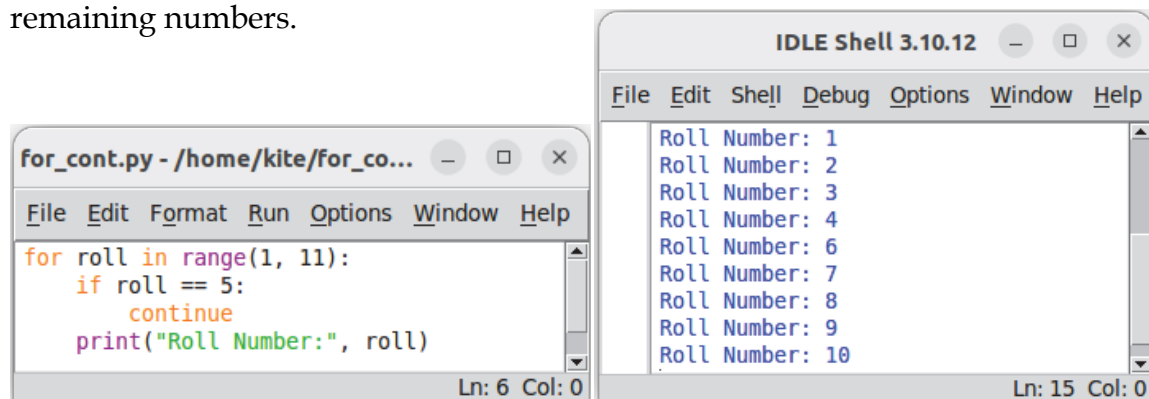
During each iteration, the program prints the current roll number. Immediately after printing, the loop checks whether the value of roll has reached 5. When this condition becomes True, the break statement is executed, which causes the loop to stop immediately, even though the condition `roll<=10` would normally allow more iterations. If the roll number is not 5, the statement `roll=roll+1` increments the value of roll, ensuring that the loop progresses correctly. Thus, the while loop prints roll numbers from 1 to 5 and then exits prematurely because of the break statement (see Figure 6.30(b)).

6.5.2 continue Statement

The continue statement is used to skip the rest of the statements in the current iteration and move directly to the next iteration of the loop. It does not terminate the loop entirely.

Problem 6.18: Using continue in a loop print roll numbers 1 to 10 except 5.

In the program shown in Figure 6.31, when the value of roll equals 5, the continue statement is executed. It skips the `print()` statement for that iteration and moves directly to the next roll number. Thus, roll number 5 is not printed, but the loop continues normally for the remaining numbers.



```

for_cont.py - /home/kite/for_co...
File Edit Format Run Options Window Help
for roll in range(1, 11):
    if roll == 5:
        continue
    print("Roll Number:", roll)
Ln: 6 Col: 0

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Roll Number: 1
Roll Number: 2
Roll Number: 3
Roll Number: 4
Roll Number: 6
Roll Number: 7
Roll Number: 8
Roll Number: 9
Roll Number: 10
Ln: 15 Col: 0

```

Figure 6.31: Using continue in for loop

6.5.3 pass Statement

The pass statement is a null statement that performs no action when executed. It is used when a statement is syntactically required but no operation is to be performed at that point in the program. This helps maintain the structure of a loop or conditional block during development, even if the logic is not yet written.

Problem 6.19: Program to count the number of non-empty subject names entered by the user using pass in a loop.

In the program shown in Figure 6.32, the user is asked to enter the names of five subjects using a for loop. During each iteration, the input entered by the user is stored in the variable subject.

```

*loop_pass.py - /home/kite/loop_pass.py (3.10.12)*
File Edit Format Run Options Window Help
# Program to count number of non-empty subject names
count = 0
for i in range(1, 6):
    subject = input("Enter subject name " + str(i) + ": ")
    if subject == "":
        pass # No subject entered - do nothing
    else:
        count = count + 1
    print("Number of iterations completed:", i)
print("Number of subjects entered:", count)
Ln: 13 Col: 0

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Enter subject name 1: English
Number of iterations completed: 1
Enter subject name 2:
Number of iterations completed: 2
Enter subject name 3: Physics
Number of iterations completed: 3
Enter subject name 4: Chemistry
Number of iterations completed: 4
Enter subject name 5:
Number of iterations completed: 5
Number of subjects entered: 3
Ln: 17 Col: 0

```

Figure 6.32: Using pass in for loop

The program checks whether the input is blank (`subject==""`). If the user presses Enter without typing anything, the pass statement is executed. Since pass does nothing, the program continues executing the remaining statements of the loop without increasing the count. If a non-empty subject name is entered, the else block is executed and the subject counter is increased by one. After each iteration, the program displays the message "Number of iterations completed", showing that the loop runs all five times regardless of the input. After the loop ends, the total number of non-empty subject names entered by the user is displayed.

The pass statement is used instead of continue because the intention is not to skip the remaining statements of the loop, but to indicate that no action is required when the input is empty. This ensures that all statements meant to execute in every iteration are executed properly.

6.5.4 else Clause with Loops

In Python, both for and while loops can have an optional else clause. The statements inside the else block execute only if the loop finishes all its iterations normally (i.e., the loop is not terminated by a break statement). The usage of else with for and while loops are shown below.

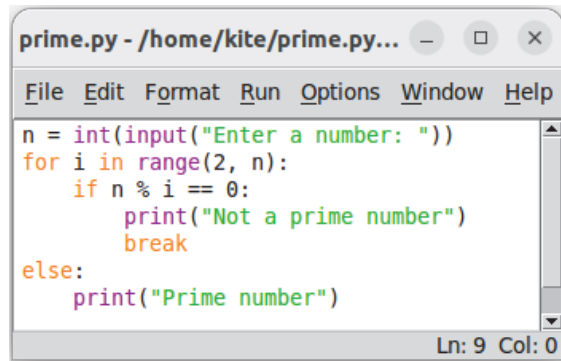
for variable in sequence:
statements
else:
statements executed
if no break occurs

while condition:
statements
else:
statements executed
if no break occurs

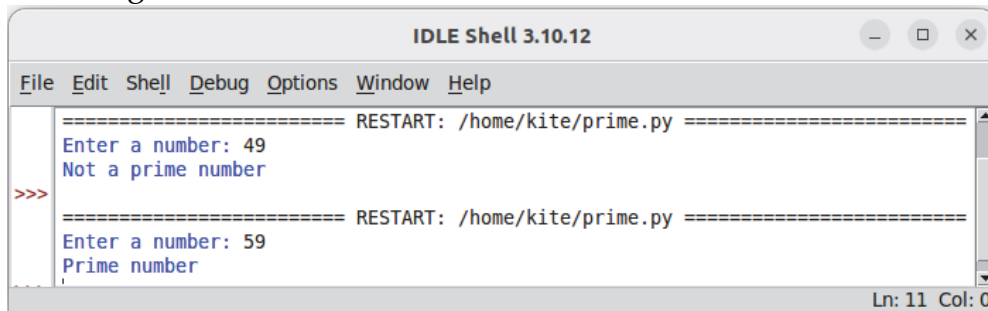
Problem 6.20: Using else with a loop to check whether the given number is prime or not.

An example of the use of else with loops is checking whether a number is prime. A prime number is greater than 1 and has no divisors other than 1 and itself. In this approach, the program tests whether the given number n is divisible by any value from 2 to $n-1$. If a divisor is found, the loop terminates immediately

using `break` and the number is declared not prime, causing the `else` block to be skipped. If the loop completes all iterations without encountering `break`, it means no divisor exists and the `else` block executes, confirming that the number is prime. This construct provides a clear and logical way to handle such conditions and improves program readability. The program and sample outputs for both cases (49 and 59) are shown in Figure 6.33.



```
prime.py - /home/kite/prime.py...
File Edit Format Run Options Window Help
n = int(input("Enter a number: "))
for i in range(2, n):
    if n % i == 0:
        print("Not a prime number")
        break
else:
    print("Prime number")
Ln: 9 Col: 0
```



```
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
===== RESTART: /home/kite/prime.py =====
Enter a number: 49
Not a prime number
>>>
===== RESTART: /home/kite/prime.py =====
Enter a number: 59
Prime number
Ln: 11 Col: 0
```

Figure 6.33: Using `else` with `for` loop to check a number prime or not



Let us do

Develop algorithms, construct flowcharts and write Python code for the following problems using Loop Control Statements (`break`, `continue`, `pass`, `else`).

Using `break`

1. A program that keeps reading numbers from the user and stops as soon as the user enters a negative number.
2. To display the numbers from 1 to 100, but stop the loop when the number 42 is reached.

Using `continue`

3. To print numbers from 1 to 30, skipping all numbers that are divisible by 3.
4. To print all roll numbers from 1 to 20, excluding the absent roll numbers given in a list (use `continue`).

Combined Keyword Applications

5. To repeatedly accept marks from the user:
 - If the user enters 999, stop the loop using `break`.
 - If the user enters a mark greater than 100, use `continue` and ignore the value.
 - Display the valid marks entered so far.
6. A PIN verification program:
 - The loop allows 5 attempts to enter a 4-digit PIN.
 - If the correct PIN is entered, print "Access Granted" and use `break`.
 - If all attempts are used without entering the correct PIN, print "Account Locked" using the `else` clause.

7. Each of the following codes contains an error (syntax, indentation, or logic), and the intended purpose of the program is given as a comment. Identify the error and rectify it.

```
1 # Purpose: Display numbers
  # from 1 to 10, but stop at 5
  for i in range(1, 11):
    print(i)
    if i == 5:
      break
```

```
2 # Purpose: Print odd
  # numbers from 1 to 10
  for i in range(1, 11):
    if i % 2 == 0:
      continue
    print(i)
```

```
3 # Purpose: Print 1 - 5
  i = 1
  while i <= 5:
    if i == 3:
      pass
    else:
      print(i)
    i = i + 1
```

8. Predict the output of the following Python codes.

```
1 i = 1
  while i <= 5:
    if i == 3:
      pass
    print(i)
    i = i + 1
```

```
2 i = 1
  while i <= 10:
    if i == 5:
      break
    print(i)
    i = i + 1
```

```
3 i = 0
  while i < 6:
    i = i + 1
    if i == 4:
      continue
    print(i)
```

```
4 for i in range(1, 8):
    if i % 2 == 0:
      continue
    print(i, end=" ")
```

```
5 for i in range(1, 10):
    if i == 7:
      break
    print(i, end=" ")
```

```
6 for i in range(1, 6):
    if i > 3:
      pass
    print(i)
```

```
7 i = 1
  while i <= 5:
    if i == 4:
      break
    print(i)
    i = i + 1
  else:
    print("Done")
```

```
8 for i in range(1, 8):
    if i == 3:
      continue
    if i == 6:
      break
    print(i)
  else:
    print("Finished")
```

```
9 i = 1
  while i <= 5:
    if i % 2 == 0:
      i = i + 1
      continue
    print(i)
    i = i + 1
  else:
    print("End")
```

Table 6.4: Summary of loop control keywords

Keyword	Purpose	Effect on Loop Execution
break	Terminates the loop immediately	Exits the loop entirely
continue	Skips the current iteration	Proceeds to the next iteration
pass	Does nothing (placeholder)	Loop continues normally
else	Executes after loop completes normally	Skipped if loop ends with break

Summary of loop control keywords are given in Table 6.4. Loop control statements in Python are used to alter the normal flow of execution of loops. While a loop usually runs repeatedly until its condition becomes False, loop control statements provide additional flexibility by allowing the program to interrupt, skip, or extend this flow when required. The `break` statement terminates the loop immediately, even if the loop condition is still True. The `continue` statement skips the remaining statements in the current iteration and

moves to the next iteration of the loop. The pass statement acts as a placeholder and performs no action, but is useful where a statement is syntactically required. The else clause with loops executes only when the loop completes normally without encountering a break. Together, these loop control statements help programmers write clearer, more efficient programs by handling exceptional situations and controlling loop behaviour precisely.

6.6 Nested Loops

Many tasks in real life involve repetition happening within another repeated activity. For example, imagine the process of entering marks of all students in all subjects during a class test:

- For each student (Roll No. 1 to 5),
- the teacher enters marks for each subject (say, 6 subjects).

Here, the outer repetition goes through students one by one, and for each student, an inner repetition goes through all their subjects. The marks of all subjects must be entered before moving to the next student. This real-life situation is an ideal example of a nested loop.

A nested loop is a loop written inside another loop. The loop that contains another loop is called the outer loop, and the loop inside it is called the inner loop. Each time the outer loop runs once,

the inner loop completes all its iterations. Nested loops are helpful when dealing with two-level repetitive tasks, such as filling tables, generating patterns, creating multiplication tables, or processing rows and columns in matrices.

6.6.1 Nested for

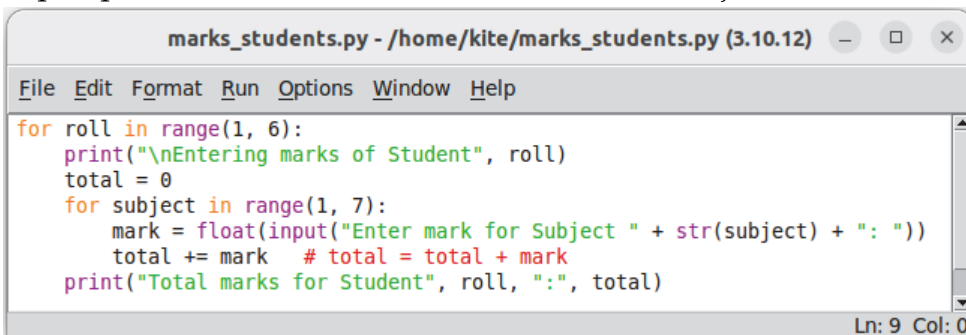
A for loop inside another one is known as nested for. General form is

```
for variable1 in sequence1:           # Outer loop
    for variable2 in sequence2:       # Inner loop
        statements
```

The use of nested for can be illustrated with examples.

Problem 6.21: Program to read marks of 6 subjects for 5 students and display the total marks of each student.

This program given in Figure 6.34(a) shows the use of a nested for loop to process the marks of 5 students across 6 subjects.



```
marks_students.py - /home/kite/marks_students.py (3.10.12)
File Edit Format Run Options Window Help
for roll in range(1, 6):
    print("\nEnter marks of Student", roll)
    total = 0
    for subject in range(1, 7):
        mark = float(input("Enter mark for Subject " + str(subject) + ": "))
        total += mark # total = total + mark
    print("Total marks for Student", roll, ":", total)
Ln: 9 Col: 0
```

Figure 6.34(a): Using nested for loop to handle multiple students data

```

*IDLE Shell 3.10.12*
File Edit Shell Debug Options Window Help

Entering marks of Student 1
Enter mark for Subject 1: 90
Enter mark for Subject 2: 89
Enter mark for Subject 3: 78
Enter mark for Subject 4: 87
Enter mark for Subject 5: 67
Enter mark for Subject 6: 65
Total marks for Student 1 : 476.0

Entering marks of Student 2
Enter mark for Subject 1: 89
Enter mark for Subject 2: 87
Enter mark for Subject 3: 68
Enter mark for Subject 4: 94
Enter mark for Subject 5: 70
Enter mark for Subject 6: 85
Total marks for Student 2 : 493.0

Entering marks of Student 3
Enter mark for Subject 1: 35
Enter mark for Subject 2: 40
Enter mark for Subject 3: 55
Enter mark for Subject 4: 30
Enter mark for Subject 5: 41
Enter mark for Subject 6: 66
Total marks for Student 3 : 267.0

Entering marks of Student 4
Enter mark for Subject 1: 45
Enter mark for Subject 2: 56
Enter mark for Subject 3: 67
Enter mark for Subject 4: 76
Enter mark for Subject 5: 65
Enter mark for Subject 6: 54
Total marks for Student 4 : 363.0

Entering marks of Student 5
Enter mark for Subject 1: |
Ln: 42 Col: 25

```

Figure 6.34(b): Partial output of program shown in Figure 6.34(a)

```

table_of_nums.py - /home/kite/...
File Edit Format Run Options Window Help

for i in range(1, 6):
    for j in range(1, 6):
        print(i, '*', j, '=', i*j)
    print()
Ln: 6 Col: 0

```

Figure 6.35(a): Using nested for loop to print Multiplication Table

The outer loop runs for each student from roll number 1 to 5. For every new student, the variable total is set to zero to ensure that marks are calculated separately. The inner loop then runs six times – once for each subject – and reads the mark entered by the user. Each mark is added to the running total inside the inner loop. After the inner loop completes for a student, the total marks of that student are displayed. In this way, the inner loop processes all subjects for one student, while the outer loop ensures that the same process is repeated for every student. This clearly illustrates how nested loops are useful when dealing with data arranged in multiple levels such as students and subjects. The output shown in Figure 6.34(b) is partial.

Problem 6.22: Displaying a 5×5 Multiplication Table

In the program shown in Figure 6.35(a), two for loops are used to generate the multiplication table from 1 to 5. The outer loop controls the value of *i*, which represents the number whose multiples are being displayed. It runs from 1 to 5. For each value of *i*, the inner loop runs through all values of *j* from 1 to 5, generating one complete set of products. Inside the inner loop, the expression *i*j* computes the product of the current values of *i* and *j*. The output is shown in Figure 6.35(b).

6.6.2 Nested while

A nested while loop uses one while loop inside another. This is useful when two levels of conditions control the repetition, such as printing patterns, validating nested input, or processing tabular data where both the number of

rows and columns depend on conditions. General form is:

```

while condition1:           # Outer loop
    while condition2:       # Inner loop
        statements

```

In this structure: The outer while loop checks its condition first. Inside it, the inner while loop runs repeatedly as long as its own condition is True. After the inner loop ends, the outer loop makes progress through its update step and repeats the process.

The use of nested while can be illustrated with an example.

Problem 6.23: Printing a right-angled triangle pattern.

In the program given (see Figure 6.36), two nested while loops are used to display a right-angled number triangle. The outer loop controls the number of rows to be printed, and the variable *i* represents the current row number. For each row, the inner loop starts with *j*=1 and prints numbers from 1 up to the current value of *i*. This results in one number in the first row, two numbers in the second row, and so on, gradually forming a triangular pattern.

Inside the inner loop, the statement `print(j, end=" ")` prints each number on the same line with a space after it. The `end=" "` argument prevents the cursor from moving to a new line after each number, ensuring that all numbers belonging to the same row appear on a single line. After the inner loop finishes printing the numbers for one row, the statement `print()` in the outer loop moves the cursor to the next line. This prepares the output area for printing the numbers of the next row. The outer loop then increases the value of *i* by 1 to begin the next row of the triangle.

Thus, the two `print()` statements serve different purposes: the first one prints the numbers horizontally in the same row, and the second one ensures proper line breaks between rows. Together, they help display the right-angled triangle neatly and clearly.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5

2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10

3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
3 * 5 = 15

4 * 1 = 4
4 * 2 = 8
4 * 3 = 12
4 * 4 = 16
4 * 5 = 20

5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25

Ln: 35 Col: 0

```

Figure 6.35(b): Multiplication tables

```

star_triangle.py - /home/kite/st...
File Edit Format Run Options Window Help
i = 1
while i <= 5:
    j = 1
    while j <= i:
        print("*", end=" ")
        j = j + 1
    print()
    i = i + 1

Ln: 10 Col: 0

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
*
* *
* * *
* * * *
* * * * *

Ln: 11 Col: 0

```

Figure 6.36: Using nested while loop to print right-angled triangle pattern

6.6.3 break and continue in Nested Loops

When loops are placed inside another loop, the statements `break` and `continue` behave with respect to the loop in which they appear. A `break` inside an inner loop stops only that inner loop and transfers control to the next statement of the outer loop, whereas a `continue` inside an inner loop skips the remaining statements of that inner loop's current iteration and moves to its next cycle. This selective control is especially useful when working with two-dimensional data, generating structured patterns, or searching within rows and columns of values. By carefully using `break` and `continue` in nested loops, programmers can control the flow of execution more precisely and handle complex looping situations efficiently. The practice of writing such programs is left to you so that you can strengthen your understanding through hands-on exploration.



Let us do

1. Write programs using nested for loops to print the following right-angled triangles based on the number of lines entered by the user.

1	1 2 3 4 5	* * * * *	1
1 2	1 2 3 4	* * * *	1 2
1 2 3	1 2 3	* * *	1 2 3
1 2 3 4	1 2	* *	1 2 3 4
	1	*	1 2 3 4 5

2. Use a nested while loop to print the multiplication tables from 1 to 10 (each row should display the table of one number).
3. Write a program using nested for loops to input the marks of 3 students in 4 subjects, and print the total marks of each student.
4. Write a program using nested while loops to print all ordered pairs (i, j) where both i and j run from 1 to 4.

Nested loops are a powerful feature in Python that allow programs to handle repetitive tasks arranged in multiple levels, such as rows and columns or groups and subgroups. Although we commonly use a `for` loop inside another `for` loop, nested loops can be created using any combination of loop types. This means we can nest a `while` loop inside a `for` loop, a `for` loop inside a `while` loop, or even a `while` loop inside another `while` loop. The choice of combination depends on whether the repetition is definite or condition-controlled. Regardless of the combination used, the basic principle remains the same: the inner loop completes all its iterations for every single iteration of the outer loop. Understanding nested loops helps in solving a wide range of problems, including pattern generation, table creation, processing two-dimensional arrays, and working with structured data efficiently.

Functions and Modules

In the previous chapter, we learned control structures such as `if`, `for`, and `while`. Using these statements, we can easily write solutions to simple problems. However, as problems grow larger, the program code also becomes longer and more complex. This makes the code harder to read, understand, and maintain. To solve this issue, it is better to break a large program into smaller, manageable parts called subprograms.

Consider a large software system such as School Management System as shown in Figure 7.1. It must handle many tasks like, admission of students, academics management, accounts and fee collection, teacher and staff management, etc. Each of these tasks is considerably large. For instance, the admission process involves receiving applications, preparing rank lists, and finally admitting students and providing admission numbers. The academic section can involve entering the marks scored in all subjects of an exam, preparing progress cards, rank lists, calculation of internal marks, etc.

While designing a software to manage such complex systems, codes that handle each major subtask is placed together in a single unit called a **module**. Modules organise code into logical, reusable units, making programs more manageable and easier to understand. This process of breaking up a large problem into smaller, simpler sub-problems is called modularisation. But these sub-problems must be further subdivided into subtasks to reduce complexity. Code for such subtasks are placed in **functions**. For instance, in the examination management module, we can have functions for calculating total marks, finding averages, issuing progress reports, etc. A function for finding total marks can be used several times in different parts of the same program. This enhances code reusability. This also reduces duplication of code and thereby reduces code size.

Python allows programmers to define their own functions and modules - known as user-defined functions and modules. In addition, Python provides several built-in functions and modules which can be used in our programs. In this chapter, we will learn how to define and use functions and modules in Python.

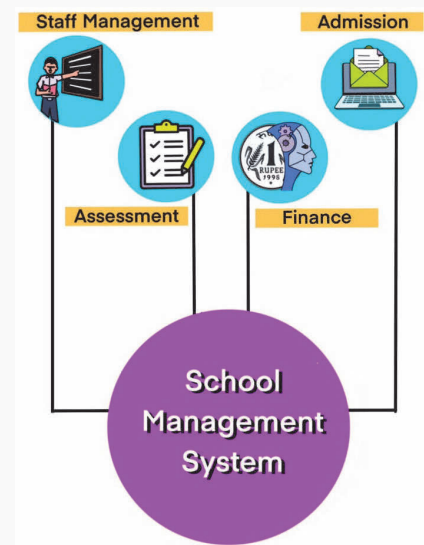


Figure 7.1:
School Management System

7.1 Functions in Python

Functions in Python are classified as built-in functions, imported functions and user-defined functions. A function receives zero or more values as inputs – called **arguments** to the function (sometimes the term parameters is also used for referring to arguments), and returns zero or more output values – called **return values** of the function. We will learn more about the use of arguments and return values as we use Python functions in the following section.

7.2 Built-in Functions

Python provides a large collection of predefined functions that are ready to be used without requiring the programmer to define them. These functions are available in Python standard library. Let us discuss some built-in functions available in Python.

a. Type Conversion Functions

Python has built-in functions that accept an input argument of one data type and returns an equivalent value in another type. These

are known as type conversion functions. Some of the type conversion functions like `int()`, `float()`, `str()` and `bool()` are discussed below.

i. **`int()`**: This function converts numeric expression to an integer value. The outputs are shown in Figure 7.2.

```

Python 3.10.12 (main, Feb  4 2025, 14:57:36) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>> int('236')
236
>>> int(121.56)
121
>>> int(-236.34)
-236
>>> int('hello')
Traceback (most recent call last):
  File "/usr/lib/python3.10/idlelib/run.py", line 578, in runcode
    exec(code, self.locals)
  File "<pyshell#3>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hello'
>>>

```

Figure 7.2: `int()` function

Input argument(s)	Return type and value	Remarks
Numeric expression as number or a string containing whole numbers.	Integer. Integer part of value of the expression is returned.	If the value given in a string contains non numeric characters, it returns an error.

ii. **`float()`**: This function converts a numeric expression given as input argument into a float type. The function and its different types of parameters are shown along with its output in Figure 7.3.

Input argument(s)	Return type and value	Remarks
Numeric expression as numbers or a string	Float. Value of the expression is returned.	If the value given in a string contains non numeric characters, it returns an error.


```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> float(34)
34.0
>>> float(34.67)
34.67
>>> float(-34.67)
-34.67
>>> float('34.67')
34.67
>>> float(34+3/4)
34.75
>>> float('34.a')
Traceback (most recent call last):
  File "/usr/lib/python3.10/idlelib/run.py", line 578, in runcode
    exec(code, self.locals)
  File "<pyshell#16>", line 1, in <module>
ValueError: could not convert string to float: '34.a'
>>>
Ln: 50 Col: 0

```

Figure 7.3: float() function

- iii. **str()**: The `str()` function converts the input into its string representation. It is shown in Figure 7.4.

Input argument	Return type and value
Any data type	String. The object is converted to string type

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> str(234)
'234'
>>> str(234.33)
'234.33'
>>> str('True')
'True'
Ln: 60 Col: 0

```

Figure 7.4: str() function

- iv. **bool()**: The `bool()` function converts a value given as input argument into its Boolean equivalent, and returns either `True` or `False`. See Figure 7.5.

Input argument	Return type and value
Numeric number, string, list, tuple, dictionary, etc.	Boolean. If the expression in the input evaluates to <code>False</code> , <code>None</code> , <code>0</code> , empty string or any empty literal collection, it returns <code>False</code> . <code>True</code> is returned otherwise.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> bool(1)
True
>>> bool(0)
False
>>> bool('')
False
>>> bool('hello')
True
Ln: 70 Col: 0

```

Figure 7.5: bool() function

b. **eval()** Function

This function takes an expression as a string as input and returns the resulting value. The working of the function is illustrated in Figure 7.6.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> eval('20+30')
50
>>> eval('20.5+30')
50.5
>>> eval('20>30')
False
>>> eval(20.5+30)
Traceback (most recent call last):
  File "/usr/lib/python3.10/idlelib/run.py", line 578, in runcode
    exec(code, self.locals)
  File "<pyshell#3>", line 1, in <module>
TypeError: eval() arg 1 must be a string, bytes or code object
Ln: 15 Col: 0

```

Figure 7.6: eval() function

Input argument	Return type and value	Remarks
A string that contains a valid Python expression.	Return type is determined by the type of the result. Result of the evaluation of the input expression is the return value.	It works with Python expressions only, not statements like if, for, while.

c. max() and min() Functions

The max() function accepts a group of values as input argument and returns the maximum value from it. The min() function accepts a group of values as input argument and returns the minimum value from it.

Note: When two string arguments are compared using max() or min(), the string that comes first in the dictionary order will be treated as having a lower value than a string that comes later in the dictionary order. However, the comparison is case sensitive. All upper-case characters will be treated to be earlier in the dictionary order than all lower-case characters. (This is because Unicode values of all upper-case characters are lower than the Unicode values of all lower-case characters.) Examples are shown in Figure 7.7.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> max(9, 23, 17, 3)
23
>>> max('Kozhikode', 'kozhikode', 'Calicut', 'calicut')
'Kozhikode'
>>> min(9, 23, 17, 3)
3
>>> min('Kozhikode', 'Wayanad', 'Kannur', 'Malappuram')
'Kannur'
Ln: 11 Col: 0

```

Figure 7.7: max(), min() functions

Input argument	Return type and value	Remarks
List, tuple, set, comma separated numbers, or comma separated strings	Return type depends on the type of the input argument. Maximum value or minimum value.	If the input elements are not comparable, then it results in an error.

d. **abs()** Function

The `abs()` function returns the absolute value of a number. See Figure 7.8.

Input argument	Return type and value	Remarks
Numeric expression	Numeric. Positive equivalent of input	If the input does not evaluate to a numeric value, it returns an error.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> abs(-50)
50
>>> abs(50)
50
>>> abs(-50.25)
50.25
>>>
Ln: 10 Col: 0

```

Figure 7.8: `abs()` function

e. **type()** Function

This function accepts a value and returns its data type. Figure 7.9 shows the different types of arguments and their data types.

Input argument	Return type and value
Variable, literals, expression, or any object in Python	Data type of the argument

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> type(10)
<class 'int'>
>>> type(23.5)
<class 'float'>
>>> type('Merin')
<class 'str'>
>>> type('23.5')
<class 'str'>
>>> type(23/5)
<class 'float'>
>>> type(True)
<class 'bool'>
>>> type(5 > 7)
<class 'bool'>
>>>
Ln: 9 Col: 5

```

Figure 7.9: `type()` function

f. **len()** Function

The `len()` function finds the number of characters in a string or the number of elements in a list, tuple etc. as shown in Figure 7.10.

Input argument	Return type and value
String, list, tuple, set or dictionary	Integer - length/number of items

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> str='Hello World'
>>> len(str)
11
>>> list=[5, 10, 20, 40]
>>> len(list)
4
>>> tuple=(23, 46, 22, 44, 55)
>>> len(tuple)
5
>>> dict={1:'Apple', 2:'Google', 3:'Oracle'}
>>> len(dict)
3
>>> set={10, 20, 30}
>>> len(set)
3
Ln: 18 Col: 0

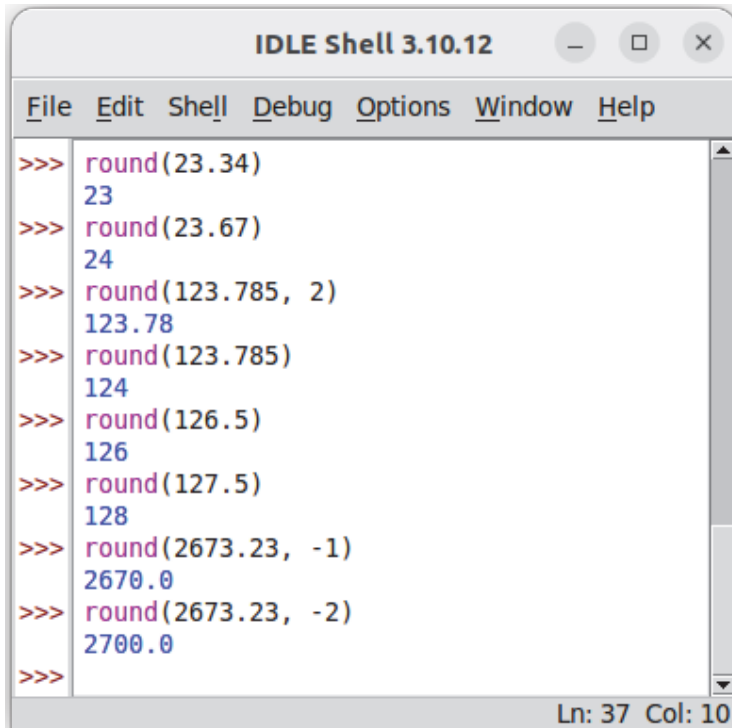
```

Figure 7.10: `len()` function

g. **round()** Function

The `round()` function rounds a given number to the specified number of decimal places. It has two input arguments – number and no. of digits. Its syntax is as given below.

`round(n, d)`



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> round(23.34)
23
>>> round(23.67)
24
>>> round(123.785, 2)
123.78
>>> round(123.785)
124
>>> round(126.5)
126
>>> round(127.5)
128
>>> round(2673.23, -1)
2670.0
>>> round(2673.23, -2)
2700.0
>>>
Ln: 37 Col: 10

```

Figure 7.11: `round()` function

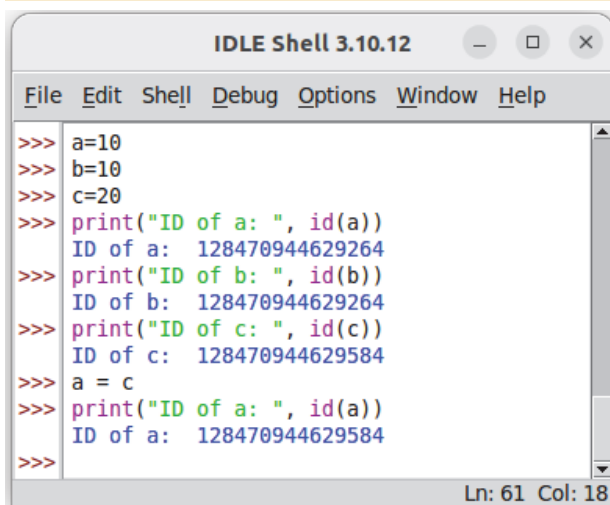
Here, **n** must be of numeric type and **d** must be an integer that specifies the number of digits upto which n is to be rounded. If d is not specified, it rounds the numbers to the nearest integer. If d is a negative number, it rounds the number upto the nearest 10^d . Note that modulus of d should be used. i.e., `round(2673.23, -2)` is rounded to 2700. This is shown in Figure 7.11.

Note: `round()` function treats the value .5 in a special way. If fractional part of a number is exactly .5 (as with 3.5 or 2.5), then it rounds it to the nearest even number (4 and 2 respectively).



Banker's Rounding

Banker's rounding, also called round half to even, is a method of rounding numbers where values exactly halfway between two integers (like 2.5, 3.5) are rounded to the nearest even integers (2, 4 respectively). This reduces the overall rounding bias in repeated calculations. Python's built-in `round()` function uses this method. In financial, scientific, and statistical calculations, rounding .5 upward always can introduce a small but consistent error. Banker's rounding balances rounding up and down, making results more accurate over large number of operations.



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> a=10
>>> b=10
>>> c=20
>>> print("ID of a: ", id(a))
ID of a: 128470944629264
>>> print("ID of b: ", id(b))
ID of b: 128470944629264
>>> print("ID of c: ", id(c))
ID of c: 128470944629584
>>> a = c
>>> print("ID of a: ", id(a))
ID of a: 128470944629584
>>>
Ln: 61 Col: 18

```

Figure 7.12: `id()` function

h. **id()** Function

The `id()` function accepts any Python variable, literal, tuple etc. as input argument and returns a unique identification number (ID) that represents the memory reference to the object. This function is often useful to check whether two variables point to the same object, as their ID values will be the same. This is illustrated in Figure 7.12.



Let us do

1. After execution of the following Python code, write the output and the datatype of the output returned.

(a) <code>int(-123)</code>	(b) <code>float("123")</code>
(c) <code>str("65")</code>	(d) <code>bool("None")</code>
2. Execute the following Python code and if it returns an error, give reasons. Otherwise give output.

(a) <code>eval("45 + True")</code>	(b) <code>eval(True)</code>
(c) <code>eval(34)</code>	(d) <code>abs(4+-7)</code>
3. Write Python code to accept a string containing comma separated integer values, convert it into numeric values and then find its length, minimum and maximum value.
4. Write Python code to accept a string containing comma separated float values, convert it into numeric values and then find its data type and its ID.

7.3 Imported Functions

Apart from the built-in functions, there are several other predefined functions in Python. All the related functions are grouped together into modules according to their purpose. For example, the `math` module contains mathematical functions like `sqrt()`, `round()`, etc. A Python module can be described as a file containing code, which can include variables, functions, and even other imported modules. These modules can be imported into our programs and we can use the functions in it. The common modules in Python are `math`, `statistics`, `string`, `random`, etc.

The keyword **import** is used to import a module into a program. The `import` command informs the Python interpreter that we will be using functions of the imported module in our program. Modules can be imported in two ways.

- a. Importing entire modules:** The entire module can be imported into a program using the syntax given below.

```
import <module name1, module name2, ...>
```

For example, the `math` module can be imported as follows.

```
import math
```

The `math` and `statistics` modules can be imported using a single statement as follows.

```
import math, statistics
```

After importing the module, we can access any function inside the imported module using the dot operator.

```
math.sqrt()
```

We can import a module and assign it a shorter or more convenient name, called an alias, using the **as** keyword.

```
import statistics as stat
```

The functions inside this module can then be imported as given below.

```
stat.mean()
```

- b. **Importing selected methods from a module:** A module can contain several methods (functions). Importing the entire module inserts all the functions in that module into our program. This increases the size of the program code on translation. Python provides the facility to import a single function or only required functions from a module to our program. The syntax for this is:

```
from <module> import <function1, function2, ...>
```

For example, the `sqrt()` function from `math` module alone can be imported to our program as follows:

```
from math import sqrt
```

The `sqrt()`, `pow()` and `cos()` functions from `math` module can be imported to our program in a single statement as follows.

```
from math import sqrt, pow, cos
```

All the functions and constants in a module can be imported using the statement:

```
from math import *
```

This is the same as `import math`.

Mathematical functions like square root, `sin`, `cos`, etc. are available in the `math` module and statistical functions like `mean`, `median`, etc. are available in the `statistics` module. Let us discuss them briefly.

7.3.1 math Module

The `math` module provides access to many mathematical functions for performing calculations such as square roots, trigonometric operations, logarithms, factorials, and more. To use them, the module must first be imported using the statement `import math`. Let us discuss some functions in this module.

- a. **`sqrt(x)`:** This function accepts a positive integer or floating point number as input argument and returns its square root. If the input argument is a negative number, it results in run time error. See Figure 7.13.

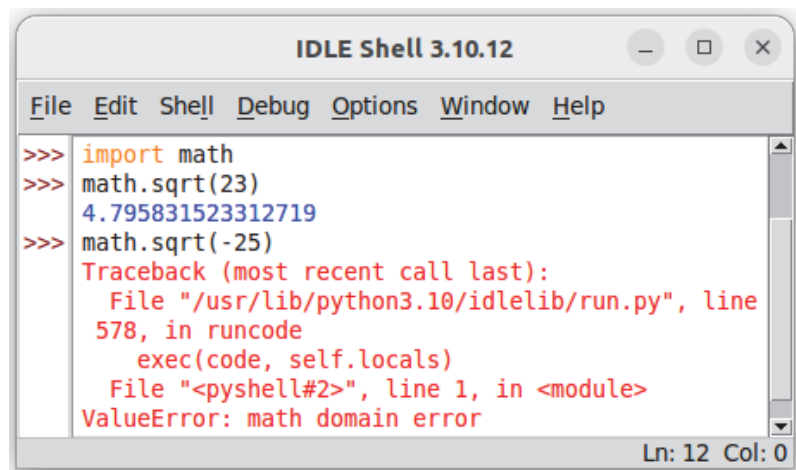
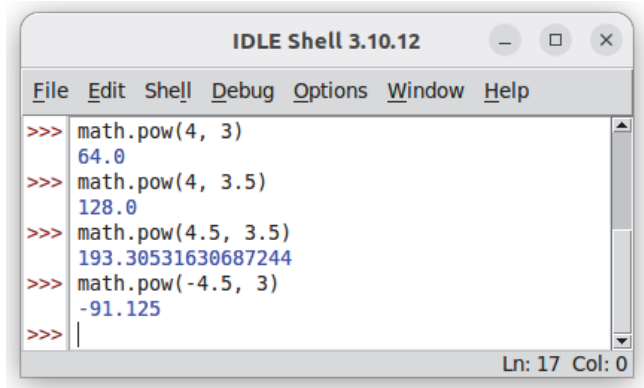


Figure 7.13: `sqrt()` function

- b. **`pow(x, y)`:** This function returns the value of x^y . It accepts base and exponent respectively as input arguments and returns the

power as shown in Figure 7.14. Base and exponent can be an integer or floating point number. The result is always a floating point number.



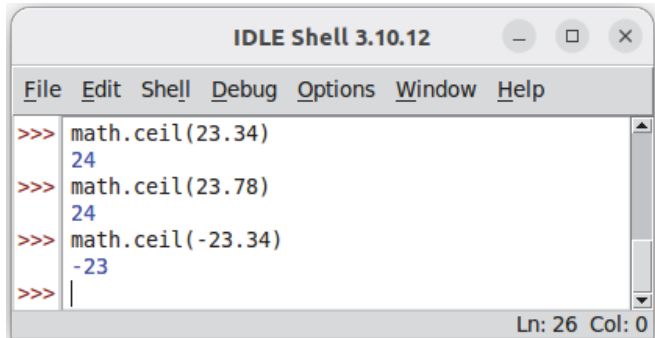
```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> math.pow(4, 3)
64.0
>>> math.pow(4, 3.5)
128.0
>>> math.pow(4.5, 3.5)
193.30531630687244
>>> math.pow(-4.5, 3)
-91.125
>>>
Ln: 17 Col: 0

```

Figure 7.14: pow() function

- c. **ceil(x)**: This function accepts an integer or floating point number as input argument and returns the closest integer that is greater than or equal to the argument as shown in Figure 7.15.



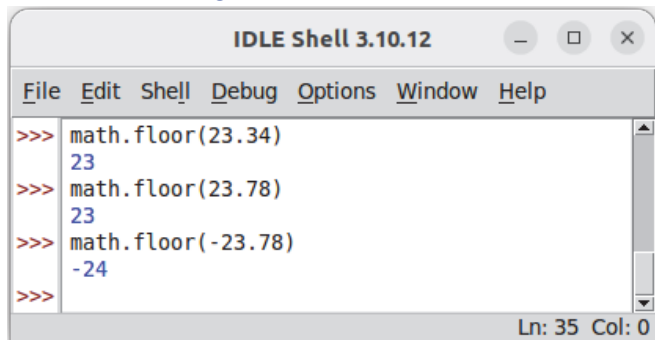
```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> math.ceil(23.34)
24
>>> math.ceil(23.78)
24
>>> math.ceil(-23.34)
-23
>>>
Ln: 26 Col: 0

```

Figure 7.15: ceil() function

- d. **floor(x)**: This function accepts an integer or floating point number as input argument and returns the closest integer that is less than or equal to the given number, as shown in Figure 7.16.



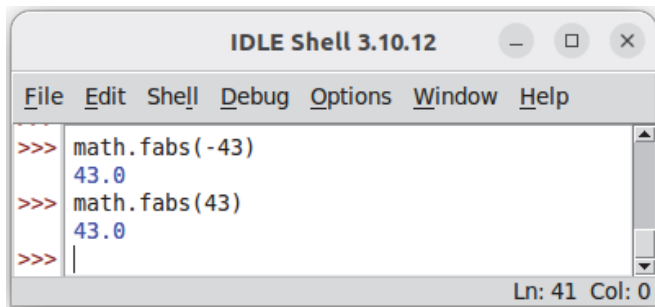
```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> math.floor(23.34)
23
>>> math.floor(23.78)
23
>>> math.floor(-23.78)
-24
>>>
Ln: 35 Col: 0

```

Figure 7.16: floor() function

- e. **fabs(x)**: This function accepts an integer or floating point number as input argument and returns its absolute value (positive value). See Figure 7.17.



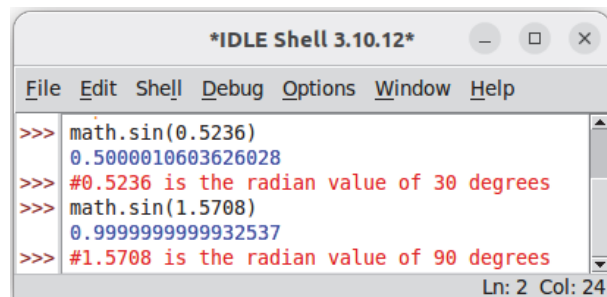
```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> math.fabs(-43)
43.0
>>> math.fabs(43)
43.0
>>>
Ln: 41 Col: 0

```

Figure 7.17: fabs() function

- f. **sin(x)**: This function accepts an integer or floating point number as input argument and returns sine value of the given number in radians. See Figure 7.18.



```

*IDLE Shell 3.10.12*
File Edit Shell Debug Options Window Help
>>> math.sin(0.5236)
0.5000010603626028
>>> #0.5236 is the radian value of 30 degrees
math.sin(1.5708)
0.999999999932537
>>> #1.5708 is the radian value of 90 degrees
Ln: 2 Col: 24

```

Figure 7.18: sin() function

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> math.cos(1.0472)
0.4999978792725457
>>> #1.0472 is the radian value of 60 degrees
>>> math.cos(2.0944)
-0.5000042414459136
>>> #2.0944 is the radian value of 120 degrees
Ln: 2 Col: 41

```

Figure 7.19: `cos()` function

- g. **`cos(x)`**: This function accepts an integer or floating point number as input argument and returns cosine value of the given number in radians, as shown in Figure 7.19.

7.3.2 statistics Module

The statistics module provides functions for performing mathematical statistics on numerical data. It is useful for calculating values such as the mean, median, mode, standard deviation, variance, etc. This module is part of Python's standard library and it does not require any external installation. It can be imported using the statement: `import statistics`.

The functions inside statistics module are designed to work with a group of values. In Python, a group of values can be a list, tuple, set, etc.

The statistics module is useful for programmers and data analysts to carry out basic statistical computations, data analysis, machine learning, and scientific applications.

- a. **`mean(data)`**: This function returns the arithmetic mean (average) of the numeric data given as input argument, as shown in Figure 7.20. This numeric data can also be a list, tuple, set or sequence of integer or float type.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> import statistics as stat
>>> stat.mean([1, 4, 6, 7, 9, 11])
6.333333333333333
>>> stat.median([1, 4, 6, 7, 9, 11])
6.5
>>> stat.mode([1, 4, 6, 7, 4, 11])
4
>>> stat.variance([1, 4, 6, 7, 4, 11])
11.5
>>> stat.stdev([1, 4, 6, 7, 4, 11])
3.391164991562634
>>>
Ln: 1 Col: 16

```

Figure 7.20: Functions in statistics module

- b. **`median(data)`**: This function returns the median (middle value) of the numeric data given as input argument. This numeric data can also be a list, tuple, set or sequence of integer or float type. See Figure 7.20.

- c. **`mode(data)`**: This function returns the mode (central tendency) of the numeric data set given as input argument, as shown in Figure 7.20. This numeric data can also be a list, tuple, set or sequence of integer or float type.

- d. **variance(data)**: This function is used to calculate the sample variance of a given set of numeric data given as input argument. This numeric data can also be a list, tuple, set or sequence of integer or float type. Variance is a measure of how far the numbers in a dataset are spread out from the mean. A large variance indicates that the data is spread out and a small variance indicates that the data is clustered closely around the mean. This is shown in Figure 7.20.
- e. **stdev(data)**: This function returns the standard deviation from a sample of data given as input argument as shown in Figure 7.20. Input data can also be a list, tuple, set or sequence of integer or float type. Standard deviation is also a measure of how spread out the numbers are. It is the square root of variance. `stdev(data)` gives results in the same units as the data, making it easier to interpret.



Let us do

Write Python code for solving the following problems and execute them on a computer to obtain results.

1. Execute the following code and write the output and return type. If it returns an error message, justify.

(a) <code>math.sqrt(-25)</code>	(b) <code>math.pow(2,3)</code>
(c) <code>math.ceil(-47.7)</code>	(d) <code>math.floor(-23.2)</code>
2. Write code to find the next higher integer of mean, median and mode of a given list of values.
3. Write code to find the mean of a given list corrected to two decimal places.
4. Write code to find the standard deviation of a list of values and find the difference of each element from standard deviation.

7.4 User-defined Functions

All the functions which we have discussed so far are predefined functions that are available in Python. A user-defined function is written to perform a specific task according to the programmer's requirements. This section explains how a programmer can create his/her own function.

7.4.1 Defining User-defined Functions

The definition of a user-defined function describes what the function does. The statements that form the body of the function are written in proper indentation. The syntax for defining a function is given below.

```
def function_name(argument_list):
    function_statement1
    function_statement2
```

Here **def** is the keyword that starts the function definition. Function name is a unique name used to identify the function. The rules for

naming identifiers which we learned in Chapter 5 are applicable to naming functions also.

The argument list (parameter list) contains the names of variables through which values are passed (given) to the function definition. You may have noticed that the predefined function `pow(x, y)` in the `math` module contains two values as arguments. These two values are processed inside the function using the instructions to find x^y . After processing, the result obtained is returned to the calling function. For that purpose, a **return** keyword is used.

Let us define a function to display a welcome message.

```
def welcome():
    print("Welcome to Python functions")
```

This function named `welcome()`, displays the message "Welcome to Python functions".

In the above example function does not return any value. The keyword **return** will be used only in functions that have atleast one value to be returned.

7.4.2 Calling User-defined Functions

The function defined above can be called as follows.

```
welcome()
```

Note that a function will be executed only when it is called. Here the output would be: Welcome to Python functions

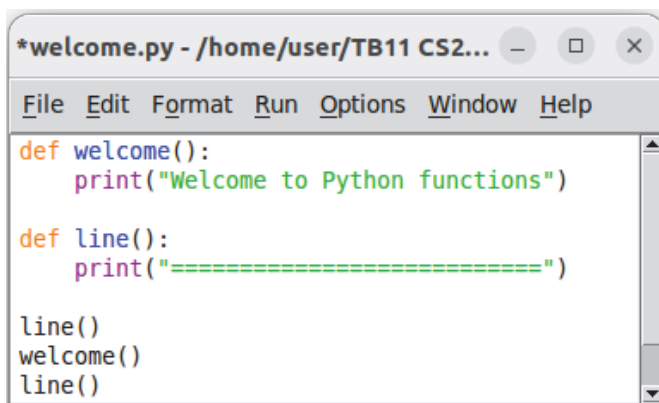


Figure 7.21(a): `welcome.py` file

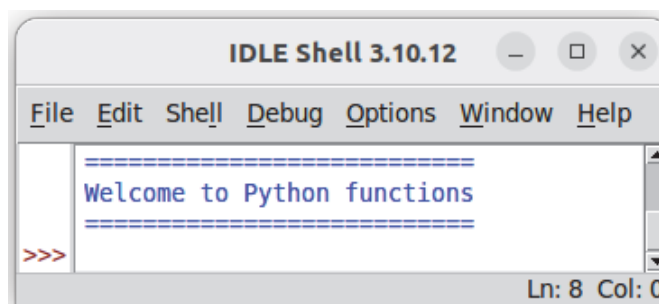


Figure 7.21(b): Output of `welcome.py`

Let us now create a file 'welcome.py' in script mode with two functions and understand the flow of execution of the program statements. The content of 'welcome.py' is displayed in Figure 7.21(a).

The order in which the statements in this file are executed is called the flow of execution. Execution always begins with the first statement of the program. The function definitions are skipped and the control moves to the function call `line()`. After executing this function, the control returns to the main function and the function `welcome()` is called and executed. The control then returns to the main function and the function `line()` is again called and executed. Note that a function must be defined before calling. The output of the code in Figure 7.21(a) is shown in Figure 7.21(b).

7.4.3 Passing Values and Returning Values

Let us now discuss how input values, called arguments (or parameters) can be passed into a function. Consider a function to calculate simple interest as given in Figure 7.22 (a).

Let us now discuss the sequence of execution of a Python program containing functions.

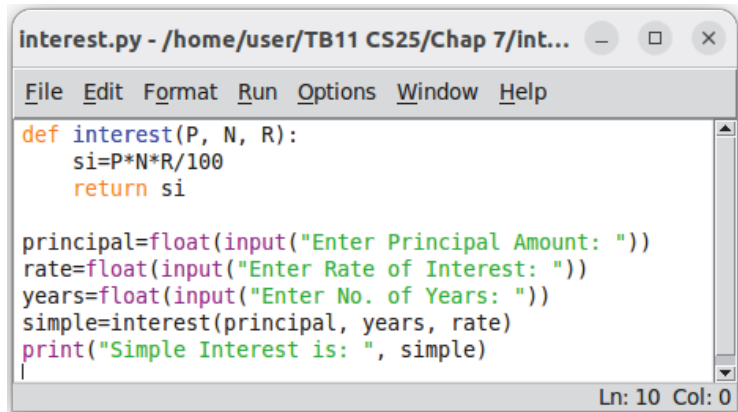
The execution starts from the first statement in the code that is outside all function/class/module definitions. Subsequently, program flows as we have already learned in Chapter 6, till a function call is encountered.

When a function call is found, control flow transfers to the first executable statement inside the function, and thereafter, execution continues inside the functions' code till either the last statement is reached or a **return** statement is encountered.

Once the execution of the function is completed (or return encountered), control shifts to the program point where the function was called, and execution proceeds in the calling program.

Python first processes all function definitions when it reads the program. It is executed only when it is called.

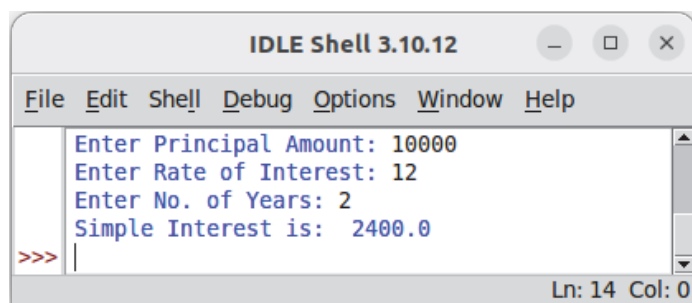
In the program discussed in Figure 7.22(a), the values in `principal`, `years` and `rate` are passed to the arguments `P`, `N` and `R` respectively when the function is called. Simple interest is calculated in the function, the value is assigned to the variable `si` and the result is returned to the function call that assigns the returned value to the variable `simple`. The output will be as shown in Figure 7.22(b).



```
def interest(P, N, R):
    si=P*N*R/100
    return si

principal=float(input("Enter Principal Amount: "))
rate=float(input("Enter Rate of Interest: "))
years=float(input("Enter No. of Years: "))
simple=interest(principal, years, rate)
print("Simple Interest is: ", simple)
```

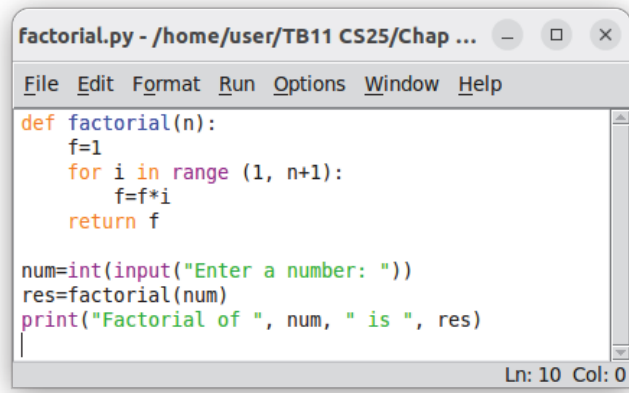
Figure 7.22(a): Function to calculate simple interest



```
Enter Principal Amount: 10000
Enter Rate of Interest: 12
Enter No. of Years: 2
Simple Interest is: 2400.0
>>>
```

Figure 7.22(b): Output of the code shown in Figure 7.22(a)

The values/variables used in the function call statement are called **actual parameters** or actual arguments. The variables used in the function definition are called **formal parameters** or formal arguments. In this example, the variables `principal`, `years` and `rate` are actual arguments and `P`, `N` and `R` are formal arguments. The **return** statement in a function definition is used to end the execution of the function and specify the value to be returned to the calling statement.



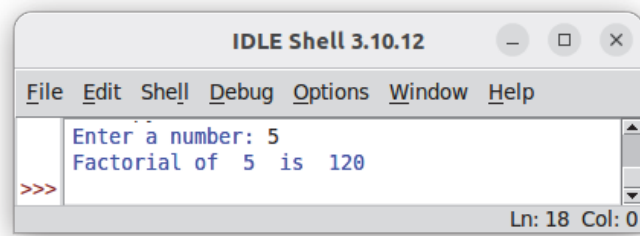
```

factorial.py - /home/user/TB11 CS25/Chap ...
File Edit Format Run Options Window Help
def factorial(n):
    f=1
    for i in range (1, n+1):
        f=f*i
    return f

num=int(input("Enter a number: "))
res=factorial(num)
print("Factorial of ", num, " is ", res)
Ln: 10 Col: 0

```

Figure 7.23(a): Function to calculate factorial



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Enter a number: 5
Factorial of 5 is 120
>>>
Ln: 18 Col: 0

```

Figure 7.23(b): Output of the `factorial()` function

Let us now consider another program to find the factorial of a number as shown in Figure 7.23(a).

If value **5** is given to the function, the output will be as shown in Figure 7.23(b).

If we pass a float value like **5.5** to the `factorial()` function, it results in an error. This is because functions in Python are typically written to operate on specific data types such as integers, floats, strings, or lists. When an argument of an incorrect type is supplied, the function may not work as intended and can cause runtime errors. In the case of the `factorial()` function, a float input leads to an error since the `range()` function used

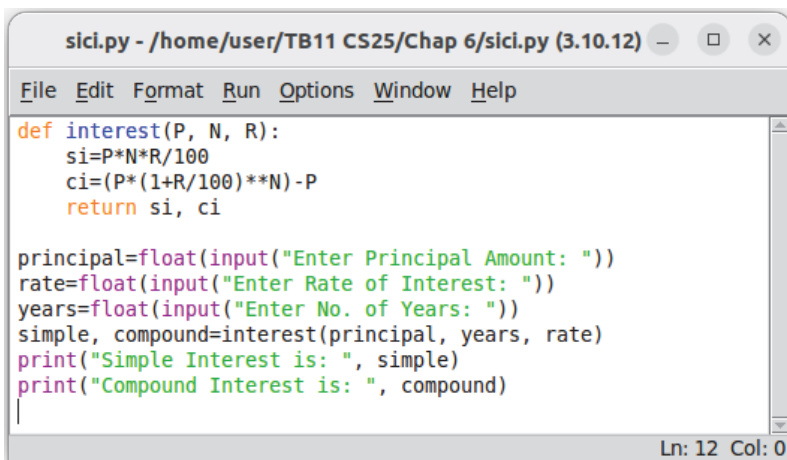
inside `factorial()` function accepts only integers. Hence, validating the type of arguments is important to avoid such unexpected errors.

In both the examples above, the `return` statement returns a single value to the calling function. Consider another example to compare simple interest and compound interest using functions. The formula

to find compound interest is $CI = (P * (1 + R / 100)^N) - P$, where `CI` is compound interest, `P` is the principal amount, `R` is rate of interest and `N` the number of years. This is shown in Figure 7.24(a).

Here `principal`, `rate` and `years` are passed to `interest()` function and the results `si` and `ci` are returned after calculation to the calling function.

When values (`si`, `ci`) are returned from a function separated by commas, Python automatically packs them into a tuple. We will learn about tuples in Chapter 10. At the call function, the tuple received is unpacked and values are copied to variables, `simple` and `compound` and then displayed as in Figure 7.24(b).



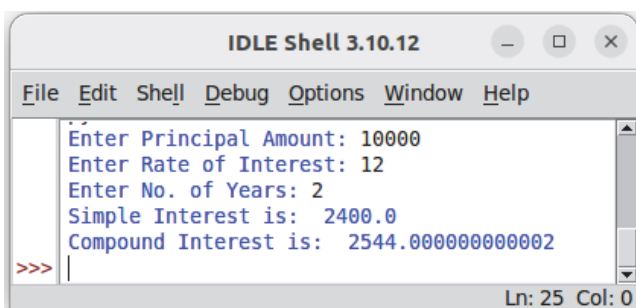
```

sici.py - /home/user/TB11 CS25/Chap 6/sici.py (3.10.12)
File Edit Format Run Options Window Help
def interest(P, N, R):
    si=P*N*R/100
    ci=(P*(1+R/100)**N)-P
    return si, ci

principal=float(input("Enter Principal Amount: "))
rate=float(input("Enter Rate of Interest: "))
years=float(input("Enter No. of Years: "))
simple, compound=interest(principal, years, rate)
print("Simple Interest is: ", simple)
print("Compound Interest is: ", compound)
Ln: 12 Col: 0

```

Figure 7.24(a): Function to calculate interest that returns multiple values



```

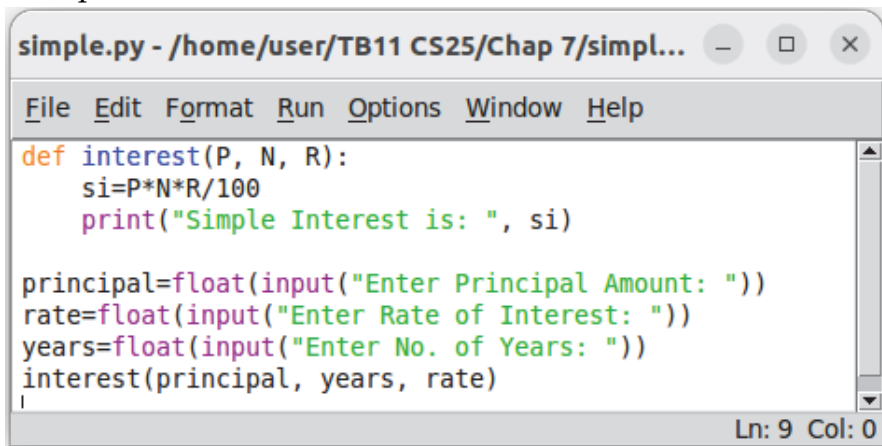
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Enter Principal Amount: 10000
Enter Rate of Interest: 12
Enter No. of Years: 2
Simple Interest is: 2400.0
Compound Interest is: 2544.0000000000002
>>>
Ln: 25 Col: 0

```

Figure 7.24(b): Output of the code given in Figure 7.24(a)

The default return value of a Python function that does not use a return statement or uses a return statement without a value is None. The keyword **None** represents the absence of a value or a null value. Let us discuss the case of functions that do not return any value.

The values are processed in the **interest()** function in Figure 7.25 and the result is displayed inside the function. This function does not return any value.



```
def interest(P, N, R):
    si=P*N*R/100
    print("Simple Interest is: ", si)

principal=float(input("Enter Principal Amount: "))
rate=float(input("Enter Rate of Interest: "))
years=float(input("Enter No. of Years: "))
interest(principal, years, rate)
```

Figure 7.25: Function to calculate interest that does not return a value

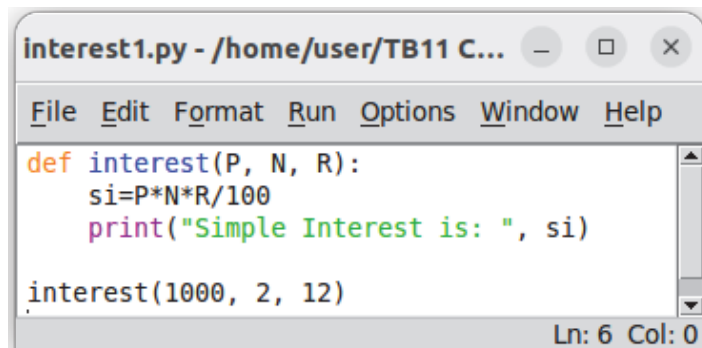
7.5 Types of Arguments

The different types of arguments available in Python are positional arguments, default arguments, keyword arguments, and variable length arguments. Let us discuss them.

7.5.1 Positional Arguments

Positional arguments are the arguments that are passed to the function definition in the correct order (or position) as in its argument list. Consider the example to find simple interest as given in Figure 7.26.

Here the order of the actual arguments in function call statement should be in the same order of formal arguments in function definition.



```
def interest(P, N, R):
    si=P*N*R/100
    print("Simple Interest is: ", si)

interest(1000, 2, 12)
```

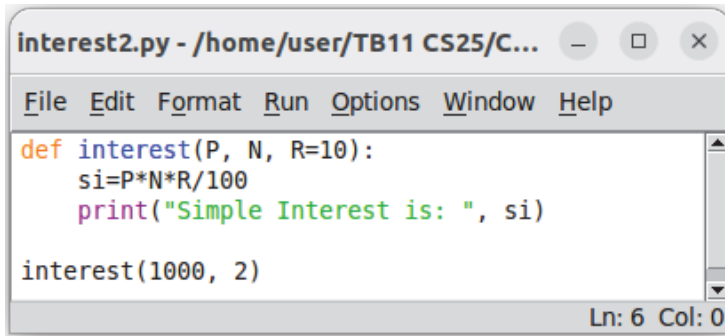
Figure 7.26: Positional arguments

7.5.2 Default Arguments

Some arguments can be given a default value. This is useful in case a value is not given for an argument in the function call statement. A value can be given to that argument in the function definition using assignment operator (=). Such arguments are called default arguments. Note that default arguments must be the right most arguments in the function definition.

Consider the function **interest()** in Figure 7.27. Here the argument **R** is set as default argument with a value 10. Therefore, if the value is not mentioned in the calling function, it will take the value 10. Consider the following function call:

```
interest(1000, 2)
```



```

interest2.py - /home/user/TB11 CS25/C...
File Edit Format Run Options Window Help
def interest(P, N, R=10):
    si=P*N*R/100
    print("Simple Interest is: ", si)

interest(1000, 2)
Ln: 6 Col: 0

```

Figure 7.27: Default arguments

is because, if a value is given for the default variable, this value will be used in the function instead of the default value given.

7.5.3 Keyword Arguments

Keyword arguments are arguments passed to a function by explicitly naming each formal argument along with its value. Here the order of arguments does not matter, as each value is assigned to the argument name, not its position. Keyword arguments make the code more readable, and clear, when functions have many arguments.

```
interest(R=12, P=1000, N=2)
```

Note that if there are any positional arguments in the function call, they should be aligned to the left side of the argument list as given below.

```
interest(1000, R=12, N=2)
```

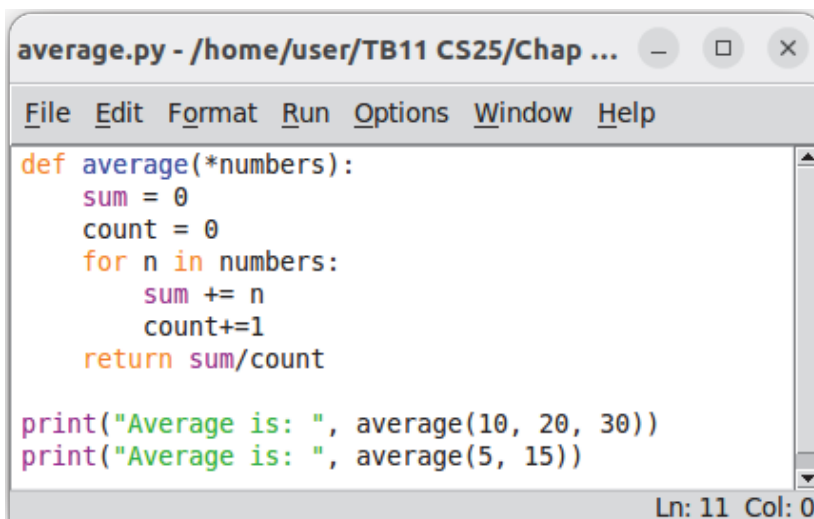
Note that the `print()` function given below, discussed in Chapter 5, can be considered as an example of keyword arguments.

```
print(value [, sep='', end='\n'])
```

Here `sep` and `end` are keyword arguments.

7.5.4 Variable Length Arguments

Variable length arguments allow a function to accept any number of arguments, making it more flexible and adaptable for different situations.



```

average.py - /home/user/TB11 CS25/Chap ...
File Edit Format Run Options Window Help
def average(*numbers):
    sum = 0
    count = 0
    for n in numbers:
        sum += n
        count+=1
    return sum/count

print("Average is: ", average(10, 20, 30))
print("Average is: ", average(5, 15))
Ln: 11 Col: 0

```

Figure 7.28(a): Variable length arguments

The output will be as follows:

Simple Interest is: 200.0

If the function call statement is:

```
interest(1000, 2, 12)
```

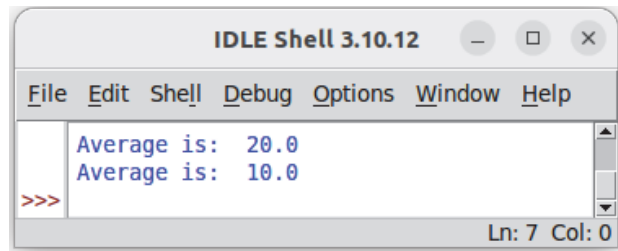
the output would be as follows:

Simple Interest is: 240.0

Here the value 12 will be used for the default argument `R`. This

The argument `*numbers` in the function `average()` in Figure 7.28(a) allows the function to accept any number of values. When we prefix an argument with `*` in the function definition, the function can accept any number of values/arguments. Here these values are stored in a tuple, named `numbers`,

internally. The output of this function will be as shown in Figure 7.28(b).



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Average is: 20.0
Average is: 10.0
>>>
Ln: 7 Col: 0

```

Figure 7.28(b): Output of the code in Figure 7.28(a)



Let us do Write Python code for solving the following problems and execute them on a computer to obtain results.

1. Write a function that accepts an integer and returns the sum of its digits.
2. Write a function that checks whether a number is prime. If it is a prime number, it returns that number and if it is not, it returns None. Using this function display the prime numbers between two integers entered by the user.
3. Write a function reverse(n) that returns the reverse of a number without converting it to a string (use modulus and floor division). After this, check whether the number is a palindrome.
4. Write a function that accepts temperature in degree Celsius and returns equivalent temperature in degree Fahrenheit. (Conversion formula: $F = (9/5 \times C) + 32$)
5. Write a function that takes a date (dd, mm, yyyy) and returns whether it is valid (True/False).
6. Write a function to input an amount in paise and convert it into rupees and paise and return them to calling function. (For example, if the input amount is 575 paise, the function should return 5 and 75 – Rs. 5 Ps. 75)
7. Write a function that accepts an integer and returns two values: number of even digits and number of odd digits.
8. Write a function that analyses a list of temperatures (readings on consecutive days) and returns the highest, the lowest, the mean temperature and the number of days above the mean.
9. Write a function that returns the first digit and the last digit of an integer.
10. Write a function that accepts an integer less than 10 and prints the number pattern as shown below for the value 4.

```

1
1 2
1 2 3
1 2 3 4

```

7.6 Scope Rules

We know that an assignment to a variable defines a reference to an object, and the variable can later be used to access the object. However, the reference remains valid only in certain parts of the program. The scope rules of Python determine the region of the program where the reference remains valid. Python stipulates that a reference assigned within a function will not remain valid outside the scope of that function – that is, the reference remains valid only for code written inside the function. References assigned outside all functions will generally remain globally valid unless overridden inside a function. These concepts are explained with examples in the following sections.

```

scope_local.py - /home/user/TB11 CS...
File Edit Format Run Options Window Help
def test():
    x = 10  # Definition of local scope.
    print(x)

test()
Ln: 6 Col: 0

```

Figure 7.29(a): Function with variable having local scope

7.6.1 Local Scope

An example of a reference that has only local scope is given in Figure 7.29(a).

Here, the variable `x` has local scope inside the function `test()` and therefore is accessible only inside this function. The output is displayed as in Figure 7.29(b).

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
10
Ln: 2 Col: 39

```

Figure 7.29(b): Output for local scope

```

scope_local_err.py - /home/user/TB1...
File Edit Format Run Options Window Help
def test():
    x = 10  # Definition of local scope.
    print(x)

test()
print(x)
Ln: 7 Col: 0

```

Figure 7.30(a): Variable `x` accessed outside function `test()`

If the variable `x` is accessed outside function `test()` as in Figure 7.30(a), it will display error as in Figure 7.30(b).

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
10
Traceback (most recent call last):
  File "/usr/lib/python3.10/idlelib/run.py", line 578, in runcode
    exec(code, self.locals)
  File "/home/user/TB11 CS25/Chap 7/scope_local_err.py", line 6, in <module>
    print(x)
NameError: name 'x' is not defined
Ln: 4 Col: 20

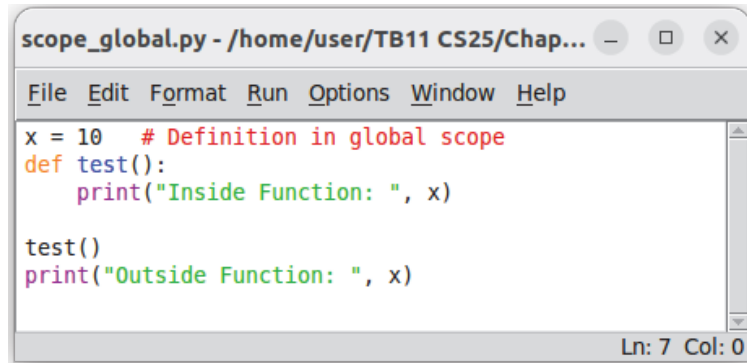
```

Figure 7.30(b): Scope error

7.6.2 Global Scope

A reference defined outside all functions has global scope. It can be accessed from any part of the program, including inside functions as

shown in Figure 7.31(a). (There is an exception to this rule, which will be discussed in the next subsection.) The output of the above program will be as shown in Figure 7.31(b).



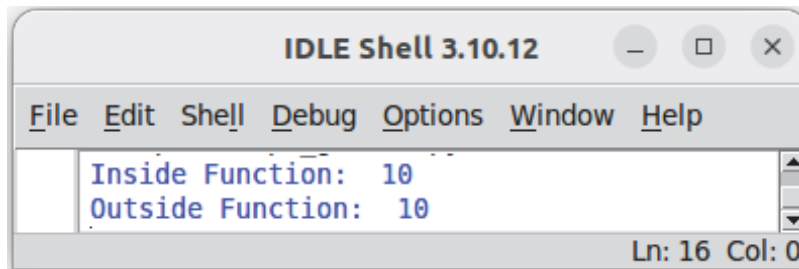
```

scope_global.py - /home/user/TB11 CS25/Chap...
File Edit Format Run Options Window Help
x = 10      # Definition in global scope
def test():
    print("Inside Function: ", x)

test()
print("Outside Function: ", x)
Ln: 7 Col: 0

```

Figure 7.31(a): Variable x with global scope



```

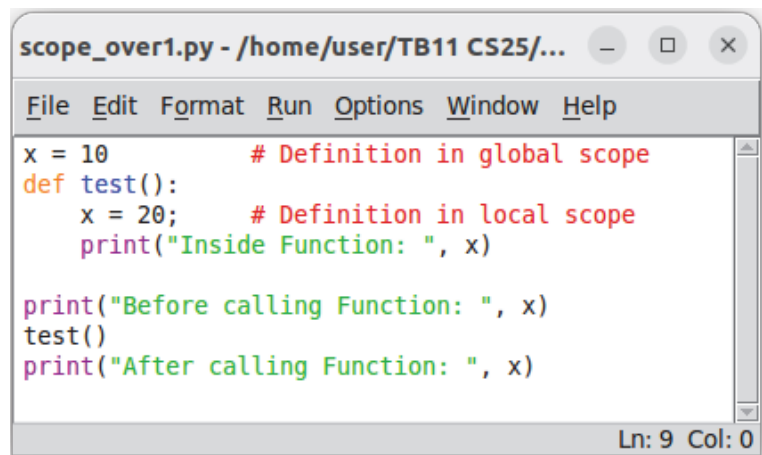
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Inside Function: 10
Outside Function: 10
Ln: 16 Col: 0

```

Figure 7.31(b): Output for global variable access

7.6.3 Overriding

The redefinition of a variable inside a function will override any global definition of the variable. That is, if a variable that was defined globally is redefined inside a function, the local definition will override the global definition. This is shown in Figure 7.32(a). The output of this program is given in Figure 7.32(b).



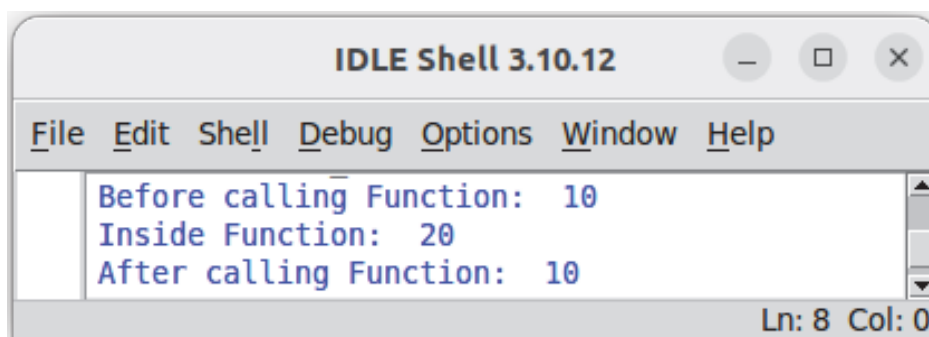
```

scope_over1.py - /home/user/TB11 CS25/...
File Edit Format Run Options Window Help
x = 10      # Definition in global scope
def test():
    x = 20;  # Definition in local scope
    print("Inside Function: ", x)

print("Before calling Function: ", x)
test()
print("After calling Function: ", x)
Ln: 9 Col: 0

```

Figure 7.32(a): Overriding of a global variable



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Before calling Function: 10
Inside Function: 20
After calling Function: 10
Ln: 8 Col: 0

```

Figure 7.32(b): Output for overriding of a global variable

To access the global variable inside a function, we need to bypass overriding of variables. Bypassing overriding of global variables means allowing a function to use and modify the global variable instead of defining a new local one with the same name. In Python,

```

scope_over2.py - /home/user/TB11 CS25...
File Edit Format Run Options Window Help
x = 10                                # Global definition
def test():
    global x                          # Global declaration
    x = x + 5
    print("Inside Function: ", x)

test()
print("Outside Function: ", x)
Ln: 9 Col: 0

```

Figure 7.33(a): Bypassing overriding of global variable x

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Inside Function: 15
Outside Function: 15
Ln: 7 Col: 0

```

Figure 7.33(b): Output for overriding of a global variable x

this is done using the keyword **global** as given in Figure 7.33(a). When global x is declared inside a function, Python refers to the x defined in the global scope. As a result, any modification made to x within the function updates the global variable. The output will be as shown in Figure 7.33(b).

7.7 Recursive Functions

Recursion is a programming technique where a function solves a problem by calling itself. Instead of working through the entire task at once, the function breaks the problem into smaller

versions of the same problem. Each recursive call works on a simpler input until it reaches a point where the problem instance becomes so simple that it can be directly solved and the value returned. This stopping condition is known as the base case of the recursive problem formulation.

Recursion is especially useful for problems that naturally repeat in smaller patterns such as computing factorials, generating Fibonacci numbers, or processing nested structures. It allows the program to express complex tasks in a clean and elegant way. A function that calls itself is called a recursive function.

A recursive function has two main parts:

Base case – It is the condition that stops the recursion. Without a base case, the function would call itself indefinitely and fail to terminate. This results in Python interpreter flagging an error.

```

rec_fact.py - /home/user/TB11 CS25/Chap 7/rec_fac...
File Edit Format Run Options Window Help
def factorial(n):
    if n < 0:
        return None          # Error case
    if (n == 1) or (n == 0):
        return 1             # base case
    else:
        return n * factorial(n - 1)  # Recursive part

print("Factorial is: ", factorial(5))
Ln: 10 Col: 0

```

Figure 7.34: Recursive function factorial()

Recursive part – The part where the function calls itself with modified arguments, bringing it closer to the terminating part (base part). Consider a program to find the factorial of a positive integer (see Figure 7.34).

The factorial of a number n is defined as:

$$n! = n * (n-1) * (n-2) * \dots * 1$$

This can be written recursively as:

$$n! = n * (n-1)!$$

The recursion terminates at:

$$1! = 1$$

Now let us discuss the working of the above program.

When the function `factorial` is called for the first time, it has value 5 for the argument n . The condition $(n==1)$ with the `if` statement inside the function is `False`. Therefore, the `else` part will be executed. It returns $5 * \text{factorial}(5-1)$,

i.e., $5 * \text{factorial}(4)$.

When we evaluate `factorial(4)` in the same way, the `else` block will be executed. It returns $4 * \text{factorial}(3)$.

Similarly, `factorial(3)` returns $3 * \text{factorial}(2)$

This in turn returns $2 * \text{factorial}(1)$

When 1 is the value for argument n , the condition $(n==1)$ is `True` and the `if` part inside the function will be executed. This will return 1 to evaluate as $2 * 1$, i.e., 2.

This will be returned to `factorial(2)`. And the above function call becomes $3 * 2$, i.e. 6. This 6 will be returned to `factorial(3)` to get $4 * 6$, i.e., 24. Again this 24 will be returned to `factorial(4)` to get $5 * 24$, i.e., 120 to evaluate the function `factorial(5)`. The flow of execution will be as shown in Figure 7.35.

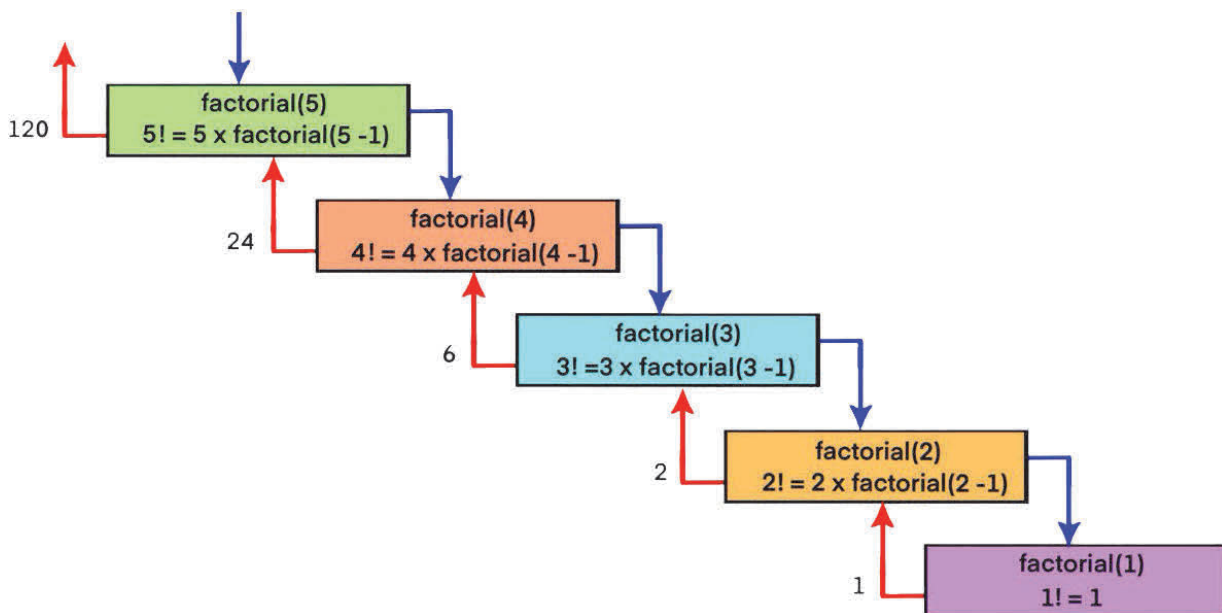
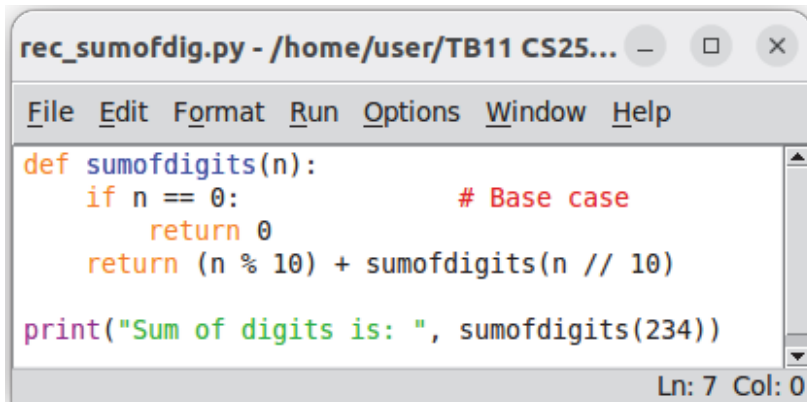


Figure 7.35: Flow of execution of `factorial(5)`

Recursion makes it easier to write and understand solutions for problems that naturally break into smaller subproblems. It reduces lengthy or complicated code into shorter, cleaner functions, improving readability. Recursion uses more memory and may be slower than loops.



```

rec_sumofdig.py - /home/user/TB11 CS25...
File Edit Format Run Options Window Help
def sumofdigits(n):
    if n == 0:                # Base case
        return 0
    return (n % 10) + sumofdigits(n // 10)

print("Sum of digits is: ", sumofdigits(234))
Ln: 7 Col: 0

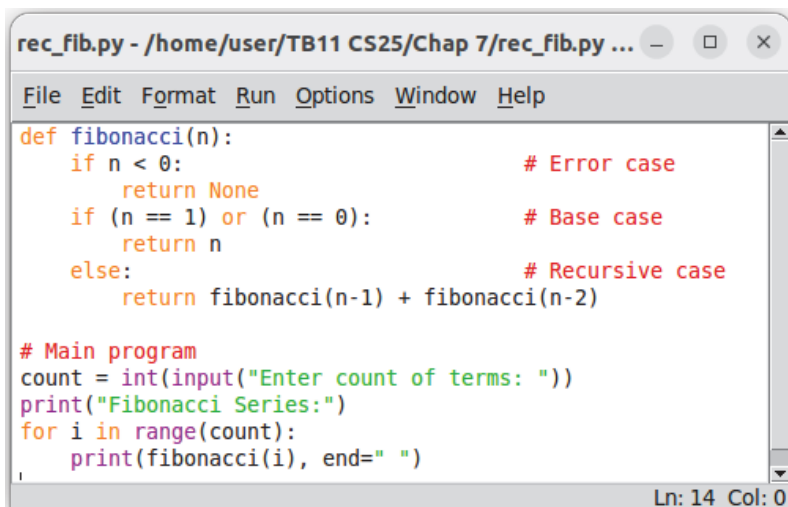
```

Figure 7.36: Recursive function - sumofdigits()

Consider another example of finding the sum of digits using recursion as shown in Figure 7.36. In this approach, the problem can be divided into two parts: the last digit of the number and the sum of the remaining digits. This continues until the remaining digit is 0 (base case).

Similarly, a recursive Python function can be used to calculate the sum of the first n natural numbers. The sum is obtained by adding the last number to the result of the recursive function applied to the previous numbers. This process continues until the number is reduced to 1 (base case). Write the recursive function for this and try it yourself using a computer.

Now consider Python code to print n Fibonacci numbers using recursive function as shown in Figure 7.37(a).



```

rec_fib.py - /home/user/TB11 CS25/Chap 7/rec_fib.py ...
File Edit Format Run Options Window Help
def fibonacci(n):
    if n < 0:                # Error case
        return None
    if (n == 1) or (n == 0):  # Base case
        return n
    else:                    # Recursive case
        return fibonacci(n-1) + fibonacci(n-2)

# Main program
count = int(input("Enter count of terms: "))
print("Fibonacci Series:")
for i in range(count):
    print(fibonacci(i), end=" ")
Ln: 14 Col: 0

```

Figure 7.37(a): Recursive function fibonacci()

The Fibonacci sequence can be computed using recursion by breaking the problem into smaller subproblems. Each term in the sequence is the sum of the two preceding terms, with the base cases defined as $\text{fibonacci}(0) = 0$ and $\text{fibonacci}(1) = 1$. In a recursive approach, the function first checks whether the input number is a base case; if not, it calls

itself to calculate the previous two Fibonacci numbers and then adds them together. This process continues, breaking the problem into smaller subproblems, until the base cases are reached. Once the base

cases return their values, the function combines all the results step by step to find the desired Fibonacci number.

Check the base case:

- If n is 0 or 1, return n .

Recursive steps:

- The base cases `fibonacci(1)` and `fibonacci(0)` evaluates to 1 and 0 respectively.
- Here first the `print()` function calls function `fibonacci(0)` and it returns 0. This value is displayed.
- Then the `print()` function calls `fibonacci(1)` is called and it returns 1, which is displayed.
- After this `fibonacci(2)` is called. This recursively calls `fibonacci(1) + fibonacci(0)`. This evaluates to 1 (1 + 0) and it is displayed.
- Similarly, `fibonacci(3)` is called which recursively calls `fibonacci(2) + fibonacci(1)`. Here `fibonacci(2)` recursively calls `fibonacci(1) + fibonacci(0)` which evaluates to 1. This means `fibonacci(3)` evaluates as 2 (1+1) which is displayed.
- Similarly `fibonacci(4)` and `fibonacci(5)` is displayed.

The output is displayed as in Figure 7.37(b).

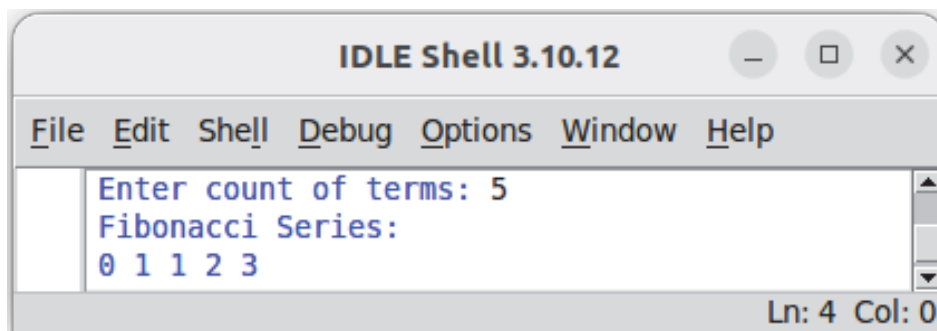


Figure 7.37(b): Output for `fibonacci()`

7.8 Creating and Using User-defined Modules

We have discussed built-in modules like `math` and `statistics` in section 7.3. Apart from using these built-in modules, we can also create our own modules to suit our specific needs. These are called user-defined modules.

A user-defined module is a Python file with file extension `.py` that contains definitions of functions, variables, or classes.

```
calculator.py - /home/user/TB11 CS25...
File Edit Format Run Options Window Help
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    return a / b
Ln: 12 Col: 0
```

Figure 7.38(a): User defined module calculator

Let us create a user defined module calculator as given in Figure 7.38(a).

Let us save the above code with the name "calculator.py". We can use this module in our program as in Figure 7.38(b).

```
calc_add.py - /home/user/TB11 CS25/C...
File Edit Format Run Options Window Help
import calculator

a = 20
b = 5
print("Addition: ", calculator.add(a, b))
Ln: 5 Col: 31
```

Figure 7.38(b): Importing calculator module to our program

The output of the above code in Figure 7.38(b) will be:

Addition: 25

```
calc_divide.py - /home/user/TB11 C...
File Edit Format Run Options Window Help
from calculator import divide

a = 20
b = 5
print("Division: ", divide(a, b))
Ln: 6 Col: 0
```

Figure 7.38(c): Importing divide() function

A specific function alone can be imported and used as in Figure 7.38(c)

The output of the above code in Figure 7.38(c) will be:

Division: 4.0

By breaking a program into modules, we can write cleaner, maintainable, and reusable code.

7.9 Hierarchical Building Blocks of Python Programs

A Python program is built using a clear hierarchy. This hierarchy helps to organise and build software, starting from small units of code and growing into larger collections of related tools. Python code is organised as functions, modules, packages, and library.

This structure helps programmers manage large projects easily by reusing and organising code. Let us discuss them.

7.9.1 Functions

A function is a block of organised and reusable code that performs a specific task. Functions help in breaking a large program into smaller parts, making it easier to write, test, and maintain. We have discussed functions in detail in this chapter.

7.9.2 Module

A module is a file that contains Python code such as functions, classes, or variables. It allows code to be logically organised and reused in different programs. A module can be imported to another program using the `import` statement. Built-in and user-defined modules are already discussed in the previous sections.

7.9.3 Package

A package is a collection of related modules grouped together in a directory/folder. It helps in organising large Python projects. A package must contain a special file named "`__init__.py`" (which can be empty) to be recognised by Python as a package. NumPy, Pandas and Matplotlib are external packages.

7.9.4 Library

A library is a collection of built-in and external modules and packages. They provide ready-to-use functionalities for various applications such as mathematics, data science, or web development. Libraries can be installed and imported like packages or modules. NumPy, Pandas, and Matplotlib are external libraries used for scientific and data analysis tasks.



Let us do

Write Python code for solving the following problems and execute them using a computer to obtain results.

1. Write a recursive function that counts the number of digits in an integer, without converting it to a string.
2. Write a recursive function to compute the power of a number (x^n) without using the `pow()` function or any loops. The function should take two arguments, `x` and `n`, where `n` is an integer. Remember that the power of a number can be calculated using the formula: $x = x * x^{n-1}$
3. Write a recursive function that returns the sum of the divisors of an integer.
4. Write a recursive function that counts the number of even digits in an integer and returns the count.
5. Write a recursive function that returns the sum of the even digits of an integer.

This chapter explained how large programs can be simplified by breaking them into smaller parts using functions and modules. It introduced different types of functions in Python, including built-in, imported, and user-defined functions. The chapter also highlighted how the use of different data types in Python affects function behaviour and output. It explained how functions receive arguments, return values, and help improve code readability and reusability. Important built-in functions such as `int()`, `float()`, `str()`, `abs()`, `len()`, and `round()` were discussed along with mathematical and statistical functions from the `math` and `statistics` modules. Students also learn how to define their own functions, use positional, default, keyword, and variable-length arguments, and return single or multiple values. The chapter concluded by explaining variable scope, recursion, user-defined modules, and the hierarchical structure of Python programs.

Strings in Python

Strings are an essential part of Python programming and widely used for storing and manipulating text data. We are already familiar with string literals from Chapter 4. In this chapter, we will learn how strings work, how to create them, how to perform various operations on string, and how to use built-in functions effectively.

A string in Python is a sequence of characters enclosed within single quotes (' '), double quotes (" "), or triple quotes (''' ''' or """ """). Strings can include letters, digits, special characters, spaces, and other characters. Python does not have a separate character type, as a character is treated as a string of length one. An individual character in a string is accessed using a index (subscript). An index is a positional value of characters in a string. The index should always be an integer (positive or negative). A index starts from 0 in Python.

The following four literals are examples of strings.

```
'Hello World '
"Python123!"
'''This is a multiline string
   with three single quotes' '''
"""This is a multiline string
   with three double quotes" """
```



Enclosing strings inside single quotes makes it easy to embed double quotes(") within the string.

Enclosing strings inside double quotes makes it easy to embed single quotes(') within the string.

Triple quotes are used for defining multiline strings or docstrings. They preserve line breaks and indentation.

8.1 String Creation

The following Python code (Figure 8.1) creates a string "Welcome to Python" and assigns it to a variable named **msg**. The `print()` function is used to display the value stored in the variable **msg**. The white space at index 7 and 10 is also counted as a character.

Table 8.1: Index of the characters of the string

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
msg	W	e	l	c	o	m	e		t	o		P	y	t	h	o	n

Strings are **immutable**, meaning once created, their contents cannot be changed directly. Any operation which modifies a string, actually creates a new string object in memory.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> msg = "Welcome to Python"
>>> print(msg)
Welcome to Python
Ln: 10 Col: 0
  
```

Figure 8.1: Creation of string

8.2 The `input()` function

The `input()` function in Python is a built-in method used to collect user input during program execution, pausing the program until the user enters data and presses Enter key. The `input()` method can be used for reading strings. As we mentioned in Chapter 5, the syntax for the `input` function is

```
var = input(prompt)
```

The `prompt` parameter is optional. If provided, it will be displayed as a message before we type the input. The input data will be stored in the variable `var`. Figure 8.2(a) is a sample code for accepting user input from the keyboard.

Figure 8.2(a): Use of `input()` method

Figure 8.2(b): Output of the above program

waits for the user input. After entering the name press Enter key. The output message will be displayed as shown in Figure 8.2(b).

8.3 Python f-strings

f-strings provide a simple and readable way to format and embed variables and expressions inside string literals. They are identified by the letter **f** or **F** placed immediately before the opening quotation mark of the string.

With f-strings, we can embed the values of variables or expressions directly into a string. The variable or expression to be embedded into a string must be placed inside curly braces `{ }` within the string itself. Python evaluates the values inside the braces at runtime and converts them to a string.

The code shown in Figure 8.3 demonstrates how to use f-strings to display a person's name and age.

In this example, the prefix `f` tells Python that this is an f-string and hence the following process takes place.

Figure 8.3: Formatted output using f - string

The variable {name} is replaced by its value ("Alice").

The variable {age} is replaced by its value (30).

We can also put simple Python expressions inside the curly braces as shown in the code in Figure 8.4(a).

```

*frmt.py - /home/chss/frmt.py (3.10.12)*
File Edit Format Run Options Window Help
price = 19.99
tax_rate = 0.08
print("Total cost without formatting", price * (1 + tax_rate))
print(f"Total cost with formatting : {price * (1 + tax_rate):.2f}")
Ln: 6 Col: 0

```

Figure 8.4(a): Use of f-string with expressions

In the above example, the prefix **f** shows that it is a Python f-string and hence the following process takes place.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Total cost without formatting 21.589199999999998
Total cost with formatting: 21.59
Ln: 7 Col: 0

```

Figure 8.4(b): Output of f-string with expressions

The expression `price * (1 + tax_rate)` is evaluated with given values as

`19.99 * (1 + 0.08)` and the result will be `21.589199999999998`.

The print statement without formatting will give the output `21.589199999999998`.

The `.2f` at the end formats the enclosed string with two decimal places and gives the output as `21.59` as displayed in Figure 8.4(a).

8.4 Accessing Characters and Indexing

Each character in a string is accessible using its index number. Positive index helps in accessing the characters of the string from the beginning. Negative subscript helps in accessing the characters of the string from the end. Index 0 or `-n` (where `n` is length of the string) refers to the first character. This is illustrated in Table 8.2. Indexing is the means to access the characters in a string. Indexing starts at 0 from the left. Negative indexing accesses characters from the right. So `-1` is the index of the last character.

Consider the string assignment statement: `text = "Hello World!"`.

Table 8.2: Indexing of strings in Python

Positive Indices	0	1	2	3	4	5	6	7	8	9	10	11
text	H	e	l	l	o		W	o	r	l	d	!
Negative Indices	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

From Table 8.2, we can understand that `text[0]` and `text[-12]` refers to the character "H" and `text[11]` and `text[-1]` refers to "!".

```

name = "Python"

print("Character at index 0 :",name[0])

print("Character at positive index 2 :",name[2])

print("Character at negative index -1:",name[-1])

```

Figure 8.5(a): Using positive and negative indices for accessing characters from string

The Python code shown in Figure 8.5(a) illustrates how characters are accessed using positive and negative indexing.

Table 8.3: Characters stored in each index of the string

Positive Indices	0	1	2	3	4	5
name	P	y	t	h	o	n
Negative Indices	-6	-5	-4	-3	-2	-1

```

Character at index 0 : P
Character at positive index 2 : t
Character at negative index -1: n

```

Figure 8.5(b): String character accessing using positive and negative indices

The output of the above code is shown in Figure 8.5(b).

In this program, `name[0]` points to the first character 'P', the positive index `name[2]` points to 't' and the negative index `name[-1]` points to the last character 'n'.

8.5 String Slicing

String slicing in Python is a method used to extract a portion (substring) of a string by specifying start, end, and optionally step indices as parameters. The parameters are included within square brackets ([]) separated by colon (:). It allows selecting specific characters or ranges of characters, which is useful for text manipulation and data extraction. The syntax to be followed for slicing is:

`string_name[start:end:step]`

where `start` is inclusive, `end` is exclusive, and `step` defines the increment. Let us have a close look at these parameters.

start The index from where the slicing begins. If omitted, slicing starts from index 0. Negative and positive values can be used as start value. This parameter is inclusive which means the character at the specified index will be selected.

end The index at which slicing stops (but the character at this index is not included). If omitted, slicing goes up to the end of the string. Negative and positive values can be used as end value. This parameter is exclusive which means the character at the specified index will be excluded.

step Specifies the interval between characters. This parameter is optional and the default value is 1. If the step value is 2 it takes every second character (skip one). A negative step value slices the string in reverse order.

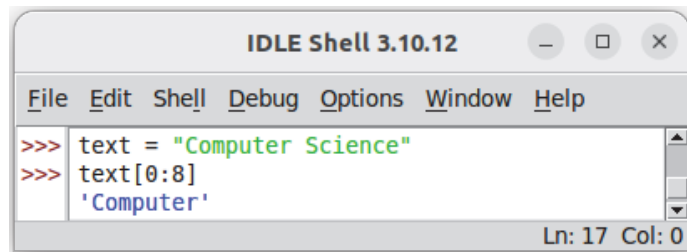
The string "Computer Science" is used here for demonstrating different ways of string slicing.

Table 8.4: Characters in each positive and negative index

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
str	C	o	m	p	u	t	e	r		S	c	i	e	n	c	e
Negative indices	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

8.5.1 Slicing Using Positive Index

Here the parameters used for specifying characters or range of characters to be extracted will be positive indices. The code given in Figure 8.6 uses positive indices.



```

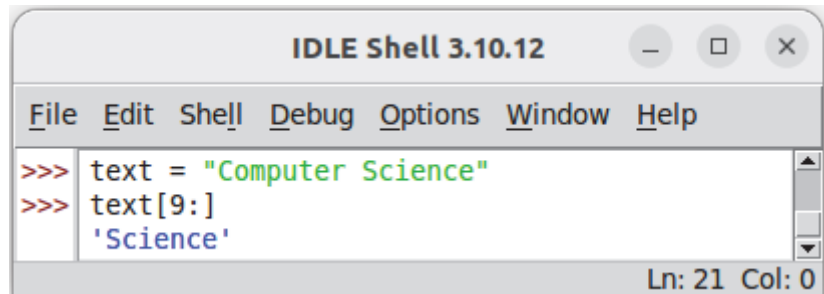
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> text = "Computer Science"
>>> text[0:8]
'Computer'
Ln: 17 Col: 0

```

Figure 8.6: Slicing using positive indices as start and stop parameters

The above code extracts a portion of the string from index 0 (start) to index 7 (end) resulting in the substring "Computer". Character at the specified index 8 will be excluded.

If the end parameter is omitted extraction starts from the specified index to the end of the string. Figure 8.7 shows slicing by specifying only the start parameter.



```

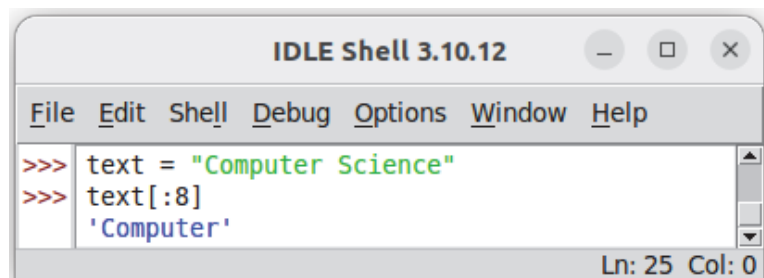
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> text = "Computer Science"
>>> text[9:]
'Science'
Ln: 21 Col: 0

```

Figure 8.7: Slicing using positive index by specifying only start parameter

In the code shown in Figure 8.7, extraction starts from index 9 (start). As the end parameter is omitted it extracts till the **end**. The resulting substring will be "Science".

If the starting index is omitted the code starts extraction from the beginning till the specified end parameter. Figure 8.8 shows slicing by specifying only the end parameter.



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> text = "Computer Science"
>>> text[:8]
'Computer'
Ln: 25 Col: 0

```

Figure 8.8: Slicing using positive index by specifying only end parameter

In the above code extraction starts from the index 0 (start) until index 7 (end) resulting in the substring "Computer". Character at the specified index 8 will be excluded. The working is same as `text[0:8]`.


```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> text = "Computer Science"
>>> text[:]
'Computer Science'
Ln: 32 Col: 0

```

Figure 8.9: Slicing without giving parameters

8.5.2 Slicing Using Negative Index

When slicing is done using negative indices, the position is counted backward from the end of the string. It will extract till the specified end parameter. Figure 8.10 shows slicing using negative indices.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> text = "Computer Science"
>>> text[-7:-1]
'Scienc'
Ln: 36 Col: 0

```

Figure 8.10: Slicing using negative parameters

In this code, as the **start** parameter is -7, slicing starts from seven characters backwards from the end. In this example slicing starts from 'S'. The last character at index -1 will be excluded. The resulting substring will be "Scienc".

If the **end** parameter is omitted as shown in Figure 8.11 characters till the end will be extracted.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> text = "Computer Science"
>>> text[-7:]
'Science'
Ln: 40 Col: 0

```

Figure 8.11: Slicing using negative parameters without specifying end parameter

In the code shown in Figure 8.11, slicing starts from seven characters backwards from the end. As the end parameter is omitted, characters till the end will be extracted.

8.5.3 Use of step in Slicing

Slicing with a **step** parameter allows us to jump or skip characters within a sequence rather than reading every single element sequentially. A positive step moves forward through the string by incrementing the index value. A negative step moves backward

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> text = "Computer Science"
>>> text[0:15:1]
'Computer Scienc'
Ln: 44 Col: 0

```

Figure 8.12: Using step parameter in string slicing

through the string by decrementing the index value. The step is an optional parameter. The default value of step parameter is 1.

Figure 8.12 shows the working of slicing using step parameter.

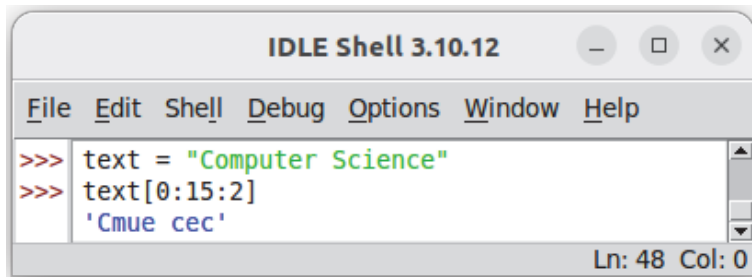
The above code extracts a substring starting from the index 0 (start) till index 14 (end) excluding the end parameter index 15. As the step parameter value is 1 no characters are skipped.

If the step value is given as 2, it skips one character and selects each second character. Figure 8.13 illustrates the working of step parameter.

In the code shown in Figure 8.13, the extraction starts from index 0 (start) and goes till index

14 (end). As the step value is given as 2 it skips one character each and select the alternate characters. This code selects the characters at indices 0, 2, 4, 6, 8, 10, 12 and 14 resulting in the substring "Cmue cec"

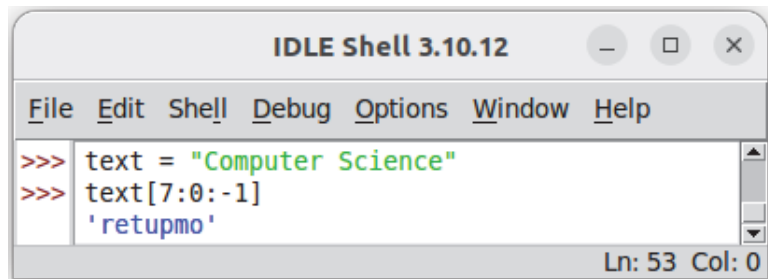
While slicing, negative step parameter value extracts a substring within the specified range in reverse order. Figure 8.14 shows how slicing is done using negative step value.



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> text = "Computer Science"
>>> text[0:15:2]
' Cmue cec '
Ln: 48 Col: 0
  
```

Figure 8.13: String slicing with step value as 2



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> text = "Computer Science"
>>> text[7:0:-1]
' retupmo '
Ln: 53 Col: 0
  
```

Figure 8.14: String slicing using negative step value

In the above code, extraction starts from the character at index 7 that is 'r'. As the step value is negative extraction will be done in reverse order. Character at the end parameter value (index 0) will be excluded. The extracted sub string will be "retupmo".



Let us do

1. Observe the string slicing operations discussed in this chapter and give justification for the output of each `print()` function.
2. Write the output of the following code.


```

string = "Python programming"
print(string[7:])
print(string[-11:-1])
print(string[-11:-1:2])
print(string[:6])
print(string[::-1])
      
```
3. Verify your answer using Python interpreter.

8.6 String Traversal

String traversal refers to the process of accessing each character in a string one by one. Since a string is a sequence of characters, Python allows us to visit or process every character using loops. When we

to traverse a string, we start from the first character and move to the next characters sequentially until we reach the last one. Traversing a string is useful when we want to perform operations like searching for a character, counting occurrences, or modifying characters. The most common way to traverse a string in Python is by using the **for** loop, which reads characters from left to right. We can also traverse using index values to access specific positions in the string.

```
traverse.py - /home/python/trav...
File Edit Format Run Options Window Help
A = "Hello"
for i in A:
    print(i)
Ln: 3 Col: 12
```

Figure 8.15(a): Illustration of string traversal

```
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
H
e
l
l
o
Ln: 153 Col: 0
```

Figure 8.15(b): Output of illustration of string traversal

The Python code and its output shown in Figure 8.15(a) illustrate string traversal operation. The string "Hello" is initially stored in a variable named **A**. When the **for** loop starts the first character at index 0 of variable **A** is accessed and assigned to variable **i**. The **print(i)** function in the body of the loop prints the first character 'H'. In the next iteration of **for** loop, the second character at index 1 of variable **A** is accessed and assigned to variable **i** and displayed. The process continues till the end of the string, accessing each element in the string. The output is shown in Figure 8.15(b).

8.7 String Operations

Python supports several operations on strings. While discussing the Python operators in Chapter 5, these operations were mentioned. Table 8.5 lists out these operations, required operator and the process involved.

Table 8.5: String operations and operators

Operation	Operator	Process
Concatenation	+	Joins two strings
Repetition	*	Repeats the string multiple times
Membership	in, not in	Checks the presence of a substring

The relational operators **>**, **<**, **>=**, **<=**, **==**, and **!=** can be used for comparing strings. The comparison is done alphabetically based on Unicode point values of characters. Comparison operations using relational operators are case sensitive. The output will be a boolean value (True / False).

Table 8.6: Relational operations on string

Operator	Description	Example	Result
==	Equal to	"apple" == "apple"	True
!=	Not equal to	"apple" != "Apple"	True
<	Alphabetically smaller	"apple" < "banana"	True
>	Alphabetically bigger	"apple" > "banana"	False
<=	Less than or equal to	"a" <= "A"	False
>=	Greater than or equal to	"curd" >= "milk"	False

Concatenation +

The + operator joins different strings and generates a new string. The Python code shown in Figure 8.16(a) and the output illustrates

```

*concat.py - /home/chss/concat.py (3.10.12)*
File Edit Format Run Options Window Help
firstname = "Guido "
lastname = "van Rossum "
info = "is known as the father of Python programming Language"
full_str = firstname + lastname + info
print(full_str)
Ln: 8 Col: 0

```

Figure 8.16(a): Python code to demonstrate string concatenation

how the operator + is used to perform concatenation operation. Figure 8.16(b) shows the output of concatenation operation.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Guido van Rossum is known as the father of Python programming Language
Ln: 71 Col: 0

```

Figure 8.16(b): Output of string concatenation

The program code combines three strings "Guido", "van Rossum" and "is known as the father of Python programming Language" into a single string "Guido van Rossum is known as the father of Python programming Language".

Repetition *

The * operator is used to repeat a string a specified number of times. The Python code and its output shown in Figure 8.17 demonstrates the string repetition operation. It prints the string "India" 3 times.

```

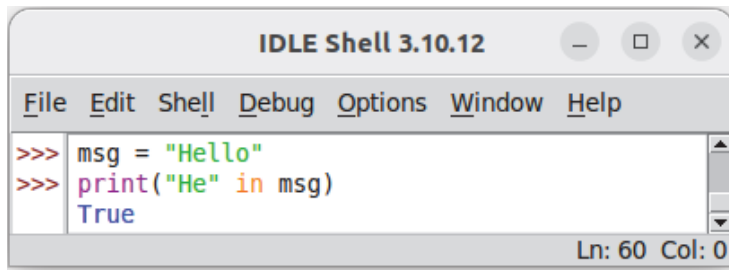
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> pattern = "India" * 3
>>> print(pattern)
IndiaIndiaIndia
Ln: 82 Col: 0

```

Figure 8.17: Python code and its output to demonstrate string repetition

Membership (in or not in)

In membership operation, the presence of a substring within another string is checked using in or not in operators. The membership operation returns a Boolean value, i.e., the output will be either True



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> msg = "Hello"
>>> print("He" in msg)
True
Ln: 60 Col: 0

```

Figure 8.18: Python code and its output to demonstrate membership operation on strings

or False. The Python code and its output given in Figure 8.18 is an example for string membership operation. The program code checks for the presence of the substring "He" in the word "Hello". The output is True as the specified substring is present in the given string.

8.7 Built-in String Functions

Python's built-in string functions are attached to string objects for tasks like formatting, searching, and validation. It returns a new string object or a boolean value. Table 8.7 shows some of those functions with its purpose and illustration.

Table 8.7: Built in functions in Python

Function	Description	Example Code	Output
capitalize()	Converts the first character of the string to uppercase.	"hello".capitalize()	'Hello'
count()	Counts the occurrences of a substring in another string	"banana".count("a")	3
find()	Finds the first index of substring. (-1 if not found.)	"apple".find("p")	1
index()	Returns the index of substring (raises error if not found).	"pineapple".index("e")	4
len()	Returns the number of characters in the string	len("Hello World")	11
replace()	Replaces a substring with another substring	'hello world'.replace('world', 'Python')	'hello Python'
startswith()	Returns True if the string starts with the specified substring	"hello".startswith("he")	True
endswith()	Returns True if the string ends with the specified substring	"hello".endswith("hi")	False

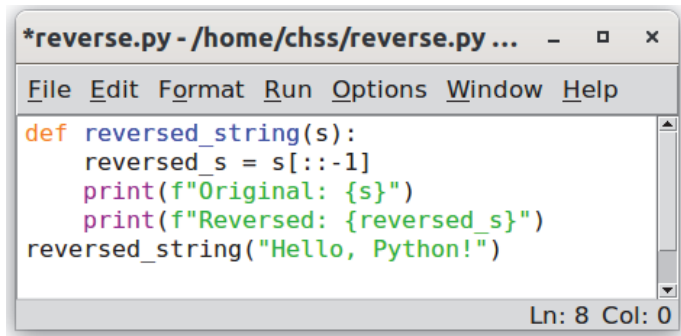
8.8 Application of Strings

In this section, we will explore how to work with strings in Python through practical examples. After understanding the concepts of

strings, various operations, and commonly used string functions, it is important to see how they are applied in real programs. The following sample programs demonstrate how to create, access, modify, and manipulate strings, along with their output for better clarity and understanding.

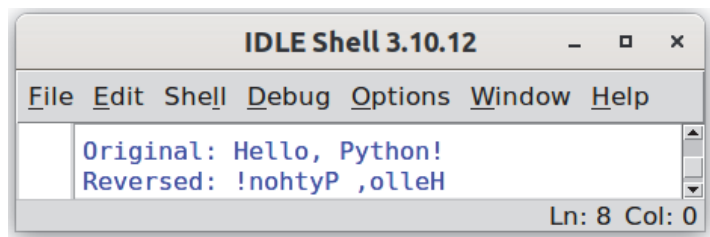
Problem 8.1: Reverse a string by writing a user-defined function. Here we use slicing operation for reversing the string. Figure 8.19(a) illustrates a Python program to reverse a string and Figure 8.19(b) shows the reversed output string.

The provided Python program defines a function called `reversed_string()` that accepts a string as input. Inside this function, the program uses a Python slicing operation (`[::-1]`) to reverse the order of characters in the input string, which is then stored in a variable named `reversed_s`. Finally, it uses `print()` statements with f-strings to display both the original string that was passed into the function and the newly created reversed string. When the program is run with the input "Hello, Python!", the output shows that the original text is displayed first, followed by the reversed text `!nohtyP ,olleH`.



```
def reversed_string(s):
    reversed_s = s[::-1]
    print(f"Original: {s}")
    print(f"Reversed: {reversed_s}")
    reversed_string("Hello, Python!")
```

Figure 8. 19(a): String reversing using user defined function and slicing method



```
Original: Hello, Python!
Reversed: !nohtyP ,olleH
```

Figure 8. 19(b): Reversed output string



Let us do

Write a Python program to check whether the given string is a palindrome.

Hint : A string is said to be palindrome if the reverse of that string is same as the original string.

After reversing the string using the above program compare it with original string. If both the strings are same print "palindrome" otherwise "not palindrome". Try with the following sample values.

Input	Output
'madam'	Palindrome
'python'	Not Palindrome
'noon'	Palindrome

Problem 8.2 : Program to count the vowels in a given string with the help of a user-defined function.

```

def count_vowels(sentence):
    vowels = "aeiouAEIOU"
    count = 0
    for char in sentence:
        if char in vowels:
            count += 1
    print(f"Sentence: '{sentence}'")
    print(f"Number of Vowels : {count}")
count_vowels("Say Yes to Life, No to Drugs")

```

Ln: 13 Col: 0

Figure 8.20(a): Finding the number of vowels in a string

The program shown in Figure 8.20(a), defines a function that counts the number of vowels in a given sentence. Inside the function, it first lists all vowels (both lower case and upper case letters) and sets a counter to zero. Then it goes through each character in the sentence one by one; whenever the character is a vowel, it adds 1 to the counter. After checking all characters, it prints the original sentence and then prints the total number of vowels it found. Here the given sentence is "Say Yes to Life, No to drugs" which contains 8 vowels.

```

Sentence: 'Say Yes to Life, No to Drugs'
Number of Vowels : 8

```

Ln: 162 Col: 0

Figure 8.20(b): Number of vowels in the string



Let us do

Write a Python program to find the number of digits in a string.

Hint : Define a function that counts how many digits are there in a string. Store the digits "0123456789" in a variable and initialise a counter to 0. Go through each character in the string and increment the counter whenever the character is a digit.

Input	Output
'C2b'	1
'April 23, 1984'	6

Problem 8.3: To find a substring within a given string and replace it with another substring. A user- defined function is designed and `replace()` function is used in it.

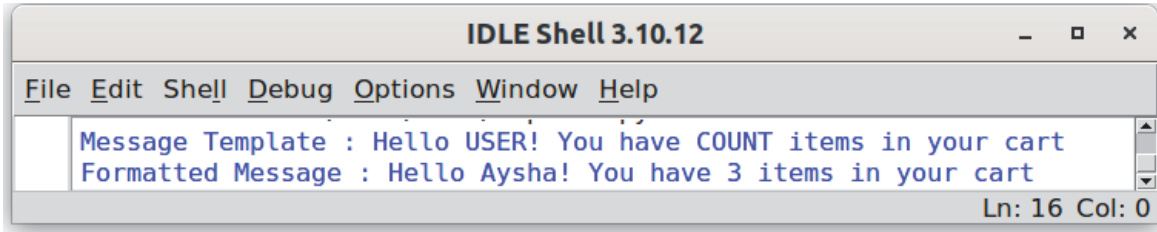
```

def format_message(name,item_count):
    message_template = "Hello USER! You have COUNT items in your cart"
    replaced_name = message_template.replace("USER", name)
    final_message = replaced_name.replace("COUNT",str(item_count))
    print(f"Message Template : {message_template}")
    print(f"Formatted Message : {final_message}")
format_message("Aysha",3)

```

Ln: 1 Col: 0

Figure 8.21(a): Python program for finding and replacing substring



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Message Template : Hello USER! You have COUNT items in your cart
Formatted Message : Hello Aysha! You have 3 items in your cart
Ln: 16 Col: 0

```

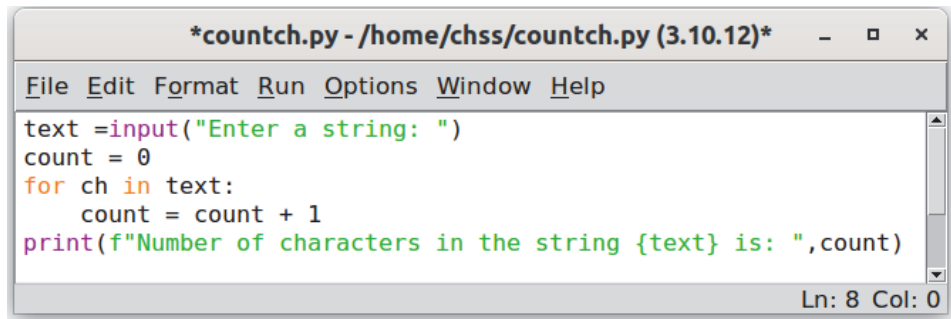
Figure 8.21(b): Replaced string output

The program given in Figure 8.21(a) defines a function to format a user message by replacing placeholders in a template string with provided name and item count values. The program defines a Python function called `format_message()` with `name` and `item_count` as arguments. A `message_template` string is initialised with placeholders "USER" and "COUNT".

The `replace()` string method is used twice to substitute these placeholders. The variable `replaced_name` stores the result after "USER" is replaced by the value of the `name` argument. The variable `final_message` stores the result after "COUNT" is replaced by the string version of the `item_count` argument (using `str()` method). The program then prints the original template and the formatted message using f-strings. Finally, the function is called with the arguments "Aysha" and 3. The output in the Python Shell shows the original template and the resulting formatted message: "Hello Aysha! You have 3 items in your cart" (refer to Figure 8.21(b)).

Problem 8.4: Print the length of a given string without using built-in functions.

In the program shown in Figure 8.22(a) the task of counting the number of characters in a string is carried out entirely



```

*countch.py - /home/chss/countch.py (3.10.12)*
File Edit Format Run Options Window Help
text =input("Enter a string: ")
count = 0
for ch in text:
    count = count + 1
print(f"Number of characters in the string {text} is: ",count)
Ln: 8 Col: 0

```

Figure 8.22(a): Python program to find number of characters in a string

through the use of a **for** loop. When the user enters a string, it is stored in the variable `text`. The variable `count` is initially set to zero and serves as the counter. The `for` loop then begins its iteration, moving through the string one character at a time. Each character in the string becomes the value of the loop variable `ch` during each cycle of the loop. With every iteration, the loop executes the statement `count = count + 1`, which increases the counter by one. This repeated action, performed once for every character, demonstrates the core idea of looping: executing the same instruction multiple times for each element in a sequence. When the loop finishes, it means all characters have been processed, and `count` now

holds the total number of characters. Finally, the program displays this value. This example clearly shows how a for loop can be used to traverse a sequence and perform cumulative operations efficiently (see Figure 8.22(b)).

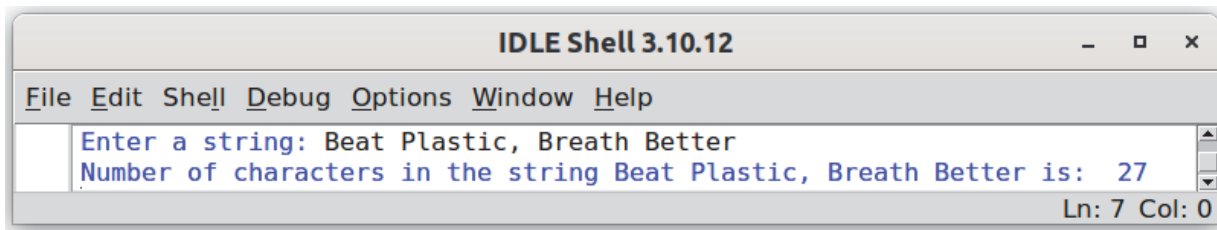


Figure 8.22(b): Number of characters in the string



Let us do

1. Write down a Python function to take two strings as input and returns as output the number of times the second string occurs as a substring of the first. Run your program and verify the output in the following test cases. (Hint: Use the functions in frequency dictionary)

Input String 1	Input String 2	Output
"she sells seashells by the seashore"	"sh"	3
"it is what it is"	"is"	2

2. In the time of Julius Caesar, messages were passed secretly between army officers by shifting each character in a message with certain shift value. For example, if the shift is 2, then "a" will be written as "c", "b" will be written as "d" and so on. If the offset is 3, then "a" will become "d", "b" will become "e" and so on. This encryption is popularly known as "Caesar Cipher".

Write a Python function to take a string and a numeric shift value as inputs and return the shifted string. Test your program with the following input.

Input String	Shift Value	Output String
"ATTACK AT DAWN"	3	"DWWDFN DW GDZQ"
"CALL ME TOMORROW"	3	"FDOO PH WRPRUURZ"

In this chapter, we have explored one of Python's most fundamental and versatile data types: strings. These sequences of characters serve as the backbone for handling text data in nearly all programming applications. We began by learning how to declare strings using single, double, or triple quotes. A crucial concept introduced was that strings are immutable—once created, their contents cannot be changed in place. This immutability ensures data integrity and affects how we approach string manipulation. We then discussed how individual characters accessed using indexing (both positive and negative) and substrings are extracted using powerful slicing techniques. The ability to iterate through a string using loops opened the door to character-by-character processing. Finally, we covered the extensive library of built-in string methods. These methods provide efficient tools for searching, modifying, and formatting text, significantly simplifying common programming tasks.

Lists

In the previous chapters, we explored how programs make decisions using control statements and how tasks can be organised using functions. These concepts help us to control the flow of a program and structure our logic. Now, we move on to the next major step in programming, handling and organising data effectively. In computer programs, data is not only stored but also needs to be arranged efficiently. The method of organising, managing, and storing data for easy access and modification is known as a **data structure**.

Data structures are collections of data in a form that makes it convenient to access and process. Programming languages provide facilities that enable programmers to define and use data structures so that data can be efficiently organised and retrieved. Python provides several built-in data structures such as lists, tuples, dictionaries, and sets. In this chapter, we will learn about these data structures and understand how they help in effective data management.

To understand the importance of organising things better, imagine a library. Books can be arranged in a single row on a shelf (linear) or grouped into sections based on subjects, authors, or languages (non-linear). In the same way, computer programs use different data structures to store and organise information depending on the needs of the task. Choosing the right data structure helps us perform operations like searching, updating, and managing large collections quickly and efficiently.

This chapter takes a look at various facilities available in Python to organise data. Python provides literal collections that allow us to group multiple values into a single unit. Among these, **lists**, the focus of this chapter, are one of the most important and flexible sequence data structures in Python. A list is an ordered collection of items that can store multiple values under a single variable. In Chapter 8, we became familiar with strings, which also maintain a sequence of characters under a single variable name. This chapter explains the structure of lists, along with concepts such as indexing, slicing, mutability, concatenation, membership testing, and commonly used list methods. It also explores advanced applications such as traversal, searching, and representing matrices using nested lists. By the end of the chapter, we will

understand how lists support real-world programming needs, from simple data storage to structured computations like matrix addition, recursion etc.

A list can hold different types of data such as numeric, boolean, or even other collections like lists, tuples, and dictionaries (which will be discussed in the coming chapters). The values stored in a list are called its **elements**. Python lists are versatile because they have the following features:

- **Ordered:** Lists are ordered sequences, meaning their elements stay in the order they are added, and each element can be retrieved using its index
- **Heterogeneous:** A list can store values of different data types.
- **Dynamic:** The size of a list can grow or shrink as needed.
- **Mutable:** Elements can be changed, added, or removed after creation.

9.1 Creating List

In Python, a list is created by enclosing its elements within square brackets `[]` and separating them with commas. Syntax for creating list is as follows :

```
list_name = [element1, element2, ...]
```

In the example given below, a list named `scores` is created.

```
>>>scores=[85, 92, 78, 96, 88]
```

Each element of the list can be identified using its corresponding index value. The indexing starts from 0. Thus, the first element has index 0, the second has index 1, and so on. In the above example, `scores` is a list containing only integers. Each integer element is separated by comma. See Figure 9.1(a) to understand how elements are arranged sequentially in the list `scores`.

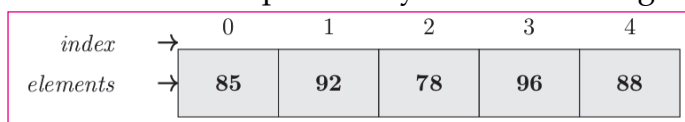


Figure 9.1(a) : Sequential arrangement of elements in the list `scores`

In Python, whenever a value is created, an object is formed in memory. We can assign different variables to these objects, and the variable names simply act as references to them. Python also keeps track of how many references point to each object, and

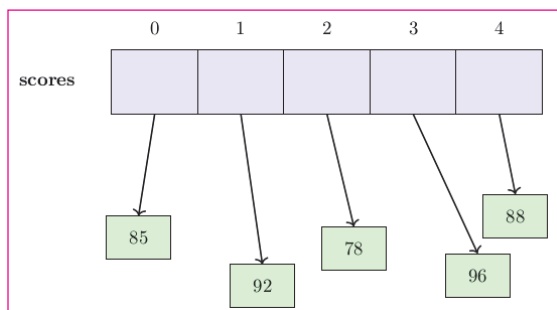


Figure 9.1(b) : Actual organisation of integer values in list

automatically removes objects that are no longer referenced. Similarly, although the diagram shows a list holding integer values directly, what is actually stored inside the list are references to those integer objects. The actual organisation of elements in lists is as depicted in Figure 9.1(b), for simplicity, we will use the pictorial representation shown in Figure 9.1(a) in the further part of this chapter.

A list can store elements of any data type. Here is an example of a list that contains strings as its elements:

```
>>>fruits = ["apple", "banana", "orange", "grape"]
```

Here, `fruits` is a list containing only strings. As we learned in the previous chapter on string handling, each string element in the list is enclosed in single/double/triple quotes. Figure 9.2 illustrates the concept of organisation of list elements in memory.

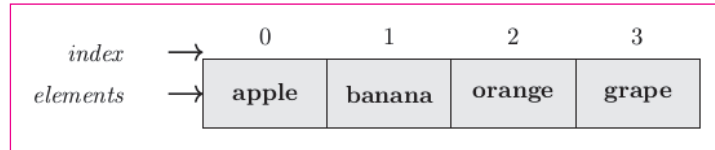


Figure 9.2: String elements are sequentially arranged in the list `fruits`



Let us do

1. Write a Python program to create the following lists:

`favourite_games` - store the names of your favourite computer games.
`favourite_books` - store the names of the books you have read.
`student_heights` - store the heights (in centimeters) of different students
 Display all three lists.

9.2 Accessing List Elements

We can access the elements of a list using their index values. The general form for accessing an element from a list is:

```
list_name[index]
```

Consider the list `fruits`, which we have created earlier. We can access its first element by using the index 0.

```
>>>print(fruits[0])
apple
```

Following code access the third element (index 2)

```
>>>print(fruits[2])
orange
```

If we try to access an index that does not exist, Python raises an `IndexError`. In the example shown in Figure 9.3, the error occurs because we attempted to access index 5 of the list `fruits`, which is beyond the range of the list we created.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> fruits = ["apple", "banana", "orange", "grape"]
>>> print(fruits[5])
Traceback (most recent call last):
  File "/usr/lib/python3.10/idlelib/run.py", line 578, in runcode
    exec(code, self.locals)
  File "<pyshell#2>", line 1, in <module>
IndexError: list index out of range
>>>
Ln: 11 Col: 0

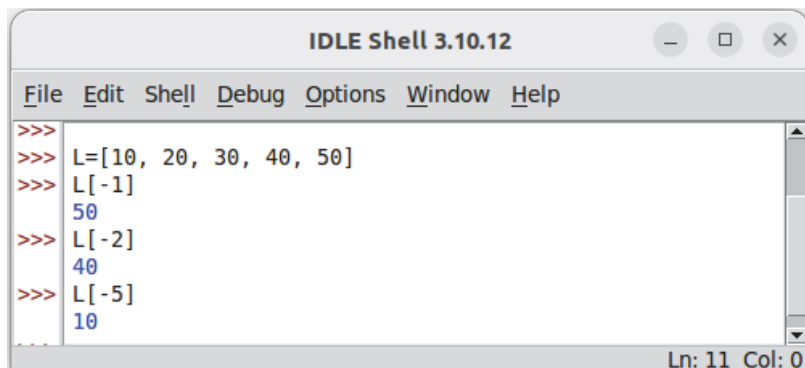
```

Figure 9.3: Invalid index access raising `IndexError`

Negative Indices

List allows the use of negative indices. A negative index indicates counting elements from the end of the list instead of the beginning. The last element has index -1, the second-last has -2, and so on. Consider the following integer list.

```
>>>L = [10,20,30,40,50]
```



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> L=[10, 20, 30, 40, 50]
>>> L[-1]
50
>>> L[-2]
40
>>> L[-5]
10
Ln: 11 Col: 0

```

Figure 9.4: Accessing list elements using negative indexing

Let us access the elements of the list using negative indexing. Various ways of using negative indices are illustrated in Figure 9.4.

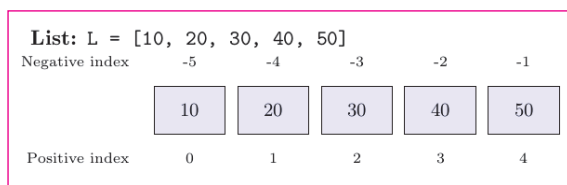


Figure 9.5: Negative and Positive indexing in list

The indexing structure corresponding to the list L is illustrated in Figure 9.5.

The use of negative indices makes it easier to access elements from the end without needing to know the list length. Various operations can be performed on lists, making them versatile for data manipulation. These built-in methods provide efficient ways to manage and process data, which is essential in programming. Let us get familiarise ourself with some of them.



Let us do

Write a Python program to create a list of equipment in your Chemistry lab, and perform the following tasks:

- Print the last item using a negative index.
- Print the second last item.
- Print the third last item.

9.3 List Slicing

As we have already seen in the previous chapter, slicing is a technique in Python that allows us to extract a specific portion of a list by specifying a range of indices. Slicing can also be applied to lists. The general form of slicing a list is:

```
list_name[start:end]
```

The slice returns elements starting from the position given by start up to, but not including, the position given by end. See the following examples based on the list colours.


```
>>>colours = ["red", "blue", "green", "yellow", "purple"]
>>>colours[1:4]
['blue', 'green', 'yellow']
```

The above code extracts elements from index 1 up to but not including index 4, resulting in the sublist ['blue', 'green', 'yellow'].

Note that in the above program, we did not use the `print()` command to display the value of the variable, however the output appears. This is a property of Python's interactive mode, which automatically displays the value of expressions.

If the start index is omitted, the slice begins from the start of the list. If the second index is omitted, the slice continues to the end of the list. When both start and end indices are omitted, the slice



```
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>>
>>> colours = ["red", "blue", "green", "yellow", "purple"]
>>> colours[:3]
['red', 'blue', 'green']
>>> colours[2:]
['green', 'yellow', 'purple']
>>> colours[:]
['red', 'blue', 'green', 'yellow', 'purple']
Ln: 11 Col: 0
```

Figure 9.6: List slicing with omitted indices

returns a new list containing all elements of the original list. These options are illustrated in Figure 9.6.

Negative indexing can also be used in slicing. Some examples are illustrated in Figure 9.7.



```
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>>
>>> colours = ["red", "blue", "green", "yellow", "purple"]
>>> colours[-3:4]
['green', 'yellow']
>>> colours[-3:]
['green', 'yellow', 'purple']
>>> colours[:-2]
['red', 'blue', 'green']
>>>
Ln: 11 Col: 0
```

Figure 9.7: Slicing with negative indices

In the code `colours[-3:4]`, the starting index -3 refers to the element "green", and the slice continues up to index 4 (not including 4) which corresponds to the element "yellow".

In the next example `colours[-3:]`, the starting index is -3 which refers to the element "green". Since the ending index is omitted, Python returns all elements from "green" to the end of the list.

We can replace a sequence of elements in a list using slicing. Look at the code snippet given below. A list L2 is created with five integers. Then the slice L2[1:4] selects the elements at positions 1, 2, and 3, that is, 10, 15, and 20. This slice is then replaced with a new list of values 100, 200 and 300.

```
>>>L2 = [5, 10, 15, 20, 25]
>>>L2[1:4] = [100,200,300]
>>>print(L2)
[5, 100, 200, 300, 25]
```

Sliced replacement works because lists are mutable, which means their elements can be changed after creation (mutability will be

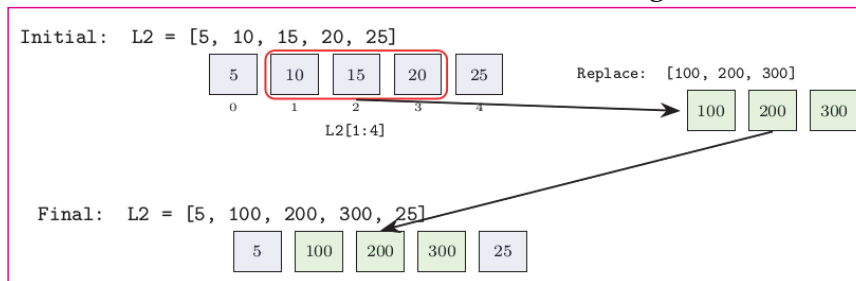


Figure 9.8: Replacing a sequence of elements in a list

explained in the next section). Therefore, Python allows us to update a specific portion of the list directly using slicing. See Figure 9.8 for illustration.

Slicing with Step Value

In Python, list slicing can include an optional third value called the **step**. The step determines how many positions to move forward or backward each time while selecting elements. The general form is:

list_name[start:end:step]

The meanings of start, end, and step are the same as we studied in string slicing:

- **start** specifies the index where the slice begins
- **end** specifies the index where the slice stops (the element at this index is not included)
- **step** specifies how many positions to move each time

Consider the following example:

```
>>>numbers=[11,12,13,14,15,16,17,18,19,20]
>>>numbers[2:7:2]
[13, 15, 17]
```

The statement selects every second element of the list numbers from index 2 to index 6. It results in [13, 15, 17], because the step size is set to 2. Python starts at index 2 (element 13) and moves forward two steps at a time, which means it skips one element between selections.

A negative step value allows slicing in the reverse direction. When the step is negative, Python starts from the start index and moves backward through the list. Using the numbers list created above, a negative step value can be applied as shown below.

```
>>>numbers[9:0:-2]
[20, 18, 16, 14, 12]
```

The slice numbers `[9:0:-2]` starts from index 9 (value 20) and moves backward with a step of -2, selecting every second element in reverse order until just after index 0, resulting in `[20, 18, 16, 14, 12]`.

A list can hold **heterogeneous values**, meaning it can store elements of different data types within the same structure. For example, the following statement is a valid list creation.

```
>>> my_bag = ["banana", 2, False, 7.25, "juice"]
```

Nested List

A list can also hold other lists as its elements. Such a structure is called a **nested list**. Nested lists help represent complex data in an organised form, such as tables, grids, or matrices, where information is arranged in rows and columns. This makes nested lists especially useful in many applications, including data science, numerical computation, and matrix-related operations. Consider the following code which creates a nested list.

```
>>>nested_list = [
    [15, 4, 19, 84],
    ["Kasaragod", "Cherthala", "Mannarkkad"],
    [2.5, 2.25, 4.2]
]
```

In the `nested_list` shown here, the first inner list contains integers, the second contains place names (strings), and the third contains floating-point values.



Let us do

Write a Python program to perform slicing operations on the following lists. Execute the program, find the output, and explain your observations about the effect of negative values in indexing.

```
numbers = [5, 10, 15, 20, 25, 30, 35, 40]
```

```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

a) `numbers[1:-2]` b) `numbers[::-2]` c) `letters[-5:-1:2]` d) `numbers[::-2]`

9.4 Mutability of List

In Python, every variable name is just a reference (or label) pointing to an object stored somewhere in memory. When we create a list, a list object is created in memory. As we learned earlier, lists in Python are mutable, which means their elements can be changed after creation. Consider the following example.

```
>>>L1 = [10, 20, 30]
```

```
>>>L2 = L1
```

Here, the variable `L1` holds a reference to a list object in memory. When we assign `L2 = L1`, both `L1` and `L2` refer to the same list object

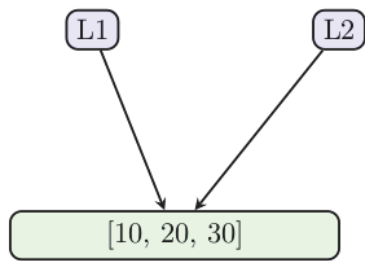


Figure 9.9: L1 and L2 points to the same list object

in memory (See Figure 9.9). When we modify the list using indexing or slice assignment, we are changing the same object in memory.

Now, let us modify the list L1 and print L2 as follows:

```
>>>L1[0] = 100
>>>print(L2)
[100, 20, 30]
```

Note that here, we modified the first element of L1. When we printed L2, the change was also reflected in it because both L1 and L2 refer to the same list object in memory. No new object was created; only the existing object was modified. This behaviour demonstrates the mutability of lists in Python. Mutability also applies to **slice assignment**. When a slice of a list is modified using assignment, the original list object is changed. However, when a slice is extracted and assigned to another variable, a new list object is created, and modifications to it do not affect the original list.

```
>>>L1 = [10, 20, 30] # Original list
>>>L2 = L1[0:2] # Slice extraction creates a new list [10, 20]
>>>L2[0] = 100 # Modification affects only L2
>>>print(L1) # L1 remains unchanged
[10, 20, 30]
>>>print(L2) # Displays the modified L2
[100, 20]
```

9.5 Concatenation of Lists

Concatenation means joining two or more lists to form a single list. Concatenation is needed when we want to combine multiple lists into one continuous list without altering the original lists. In Python, the + operator is used to concatenate lists. Consider the following example.

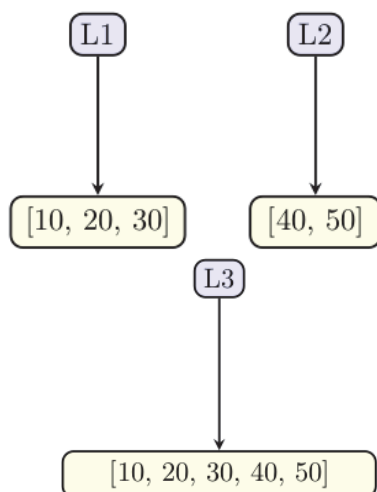


Figure 9.10: Concatenation of L1 and L2 results in a new L3 object

```
>>>L1 = [10,20,30]
>>>L2 = [40,50]
>>>L3 = L1+L2
>>>L3
[10, 20, 30, 40, 50]
```

When we use + to concatenate lists, a new list object is created, and the reference now points to a different memory location. Here, a new list object is created by concatenating L1 and L2. L3 refers to a new memory location, while L1 and L2 remain unchanged. This idea is illustrated in Figure 9.10.

Now let us see another example where two lists store string values. Consider the example given

in Figure 9.11 Two lists, classA and classB, are concatenated using the + operator. When the combined list is printed, all elements from both lists appear in the output. Note that a common element present in both lists ("Asha") appears twice in the new list because duplicates are preserved during concatenation.

```

concatenate_list.py - /home/kite/PythonWorkben...
File Edit Format Run Options Window Help
classA = ["Asha", "Binu", "Chitra"]
classB = ["Deepa", "Asha", "Fahad"]
all_students = classA + classB
print(all_students)
Ln: 4 Col: 0

File Edit Shell Debug Options Window Help
['Asha', 'Binu', 'Chitra', 'Deepa', 'Asha', 'Fahad']
Ln: 7 Col: 0

```

Figure 9.11: Program that Concatenates two lists

9.6 List Functions and Operations

Python provides several built-in functions to work with lists. These functions make it easy to add, remove, or modify elements. In this section we will understand some useful functions and operations.

9.6.1 Membership Operators

Python provides simple operators to check whether an element is present in a list. The membership operators **in** and **not in** are used to test if a value exists within a list. The **in** operator checks whether a given element is present in the list, while **not in** verifies that an element is not present. Consider the example in Figure 9.12. Here,

solar_planets is a list of strings and we are performing in operation with different string values. Observe the results: Uranus is a member of solar_planets, so the in operation returns True, whereas Pluto is not an element of solar_planets, so it returns False.

The not in operator checks if an element is absent in the list. The program given in Figure 9.13 performs not in operation with different string values. Observe the results.

```

in_operator.py - /home/kite/PythonWorkbench/in_operator.py (3.10.12)
File Edit Format Run Options Window Help
solar_planets = ["Mercury", "Venus", "Earth", "Mars",
                 "Jupiter", "Saturn", "Uranus", "Neptune"]
print("Membership of Uranus: ", "Uranus" in solar_planets)
print("Membership of Pluto: ", "Pluto" in solar_planets)
Ln: 5 Col: 0

File Edit Shell Debug Options Window Help
Membership of Uranus: True
Membership of Pluto: False
Ln: 8 Col: 0

```

9.12: Working of the in operator in a list

```

not_in.py - /home/kite/PythonWorkbenc...
File Edit Format Run Options Window Help
water_elements = ["Hydrogen", "Oxygen"]
print('Carbon' not in water_elements)
print('Oxygen' not in water_elements)
Ln: 1 Col: 0

File Edit Shell Debug Options Window Help
True
False
Ln: 8 Col: 0

```

Figure 9.13: Working of the not in operator in a list

9.6.2 More Operations

Some commonly used list methods and operations are shown in Table 9.1 along with examples. To use a method on a list, the dot operator (.) is applied after the list name. All examples in the table are based on the lists provided below.

```
>>>colours = ["red", "blue", "green", "yellow", "purple"]
>>>fibonacci = [1, 1, 2, 3, 5]
```

Table 9.1: Commonly Used List Manipulation Functions

Function/ Method	Description	Example
append()	Adds an item to the end of the list, increasing its size.	<pre>>>>fibonacci.append(8) >>>print(fibonacci) [1, 1, 2, 3, 5, 8]</pre>
insert()	Inserts an element at a specified index in the list.	<pre>>>>colours.insert(2, "black") >>>print(colours) ['red', 'blue', 'black', 'green', 'yellow', 'purple']</pre>
pop()	Removes and returns an element at a given position. If no position is specified, removes the last element.	<pre>>>>print(colours) ['red', 'blue', 'black', 'green', 'yellow', 'purple'] >>>print(colours.pop(3)) green >>>print(colours) ['red', 'blue', 'black', 'yellow', 'purple']</pre>
remove()	Removes the first occurrence of a specified element from the list.	<pre>>>>colours.remove("blue") >>>print(colours) ['red', 'black', 'yellow', 'purple']</pre>

In addition to list-specific functions discussed above, Python also provides some generic operations that work on lists. Consider the list `colours = ["red", "blue", "green", "yellow", "purple"]`.

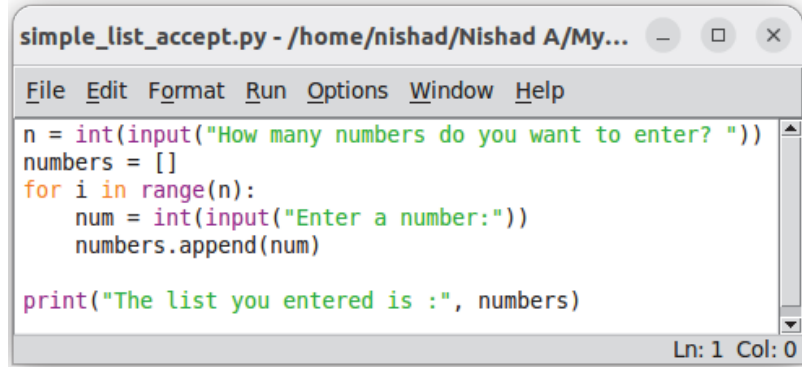
The `len()` function returns the number of elements in a list.

```
>>> print(len(colours))
5
```

The `del` statement deletes an element or a slice from the list without returning the removed value, and can also delete the entire list

```
>>> del colours[3]
>>> print(colours)
['red', 'blue', 'green', 'purple']
```


Let us look at a simple program that accepts a few integer values from the user and prints them. The program shown in Figure 9.14(a) accepts integer values from the keyboard and appends in to the list and its output is displayed in Figure 9.14(b).

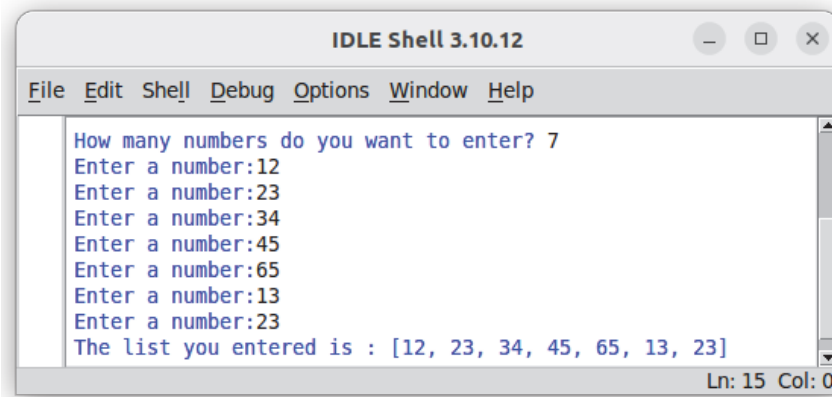


```

simple_list_accept.py - /home/nishad/Nishad A/My...
File Edit Format Run Options Window Help
n = int(input("How many numbers do you want to enter? "))
numbers = []
for i in range(n):
    num = int(input("Enter a number:"))
    numbers.append(num)
print("The list you entered is :", numbers)
Ln: 1 Col: 0

```

Figure 9.14(a) : Program to read integers into a list



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
How many numbers do you want to enter? 7
Enter a number:12
Enter a number:23
Enter a number:34
Enter a number:45
Enter a number:65
Enter a number:13
Enter a number:23
The list you entered is : [12, 23, 34, 45, 65, 13, 23]
Ln: 15 Col: 0

```

Figure 9.14(b) : Displaying the accepted list



Let us do

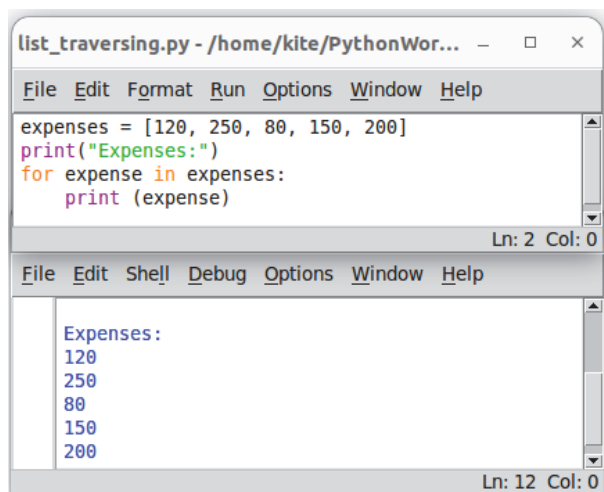
1. Consider the list `letters = ['a', 'b', 'c', 'd', 'e', 'f']` to compare the outputs of the following two slice statements and explain the reason for the difference. Execute and analyse
a) `letters[3:0:-1]` b) `letters[3:0:1]`
2. Write a Python program to print the reverse of a list using negative indexing.

9.7 Traversing List

Traversal means visiting each element of a list to perform operations such as counting, updating, or checking certain conditions. It is a fundamental skill when working with data structures. There are different ways to traverse or iterate through a list. Let us explore one commonly used method for list traversal.

9.7.1 Direct Traversal

This is the simplest way to read or process the items in a list. Consider the list `expenses`, which stores integer values representing different costs. Using a simple for loop, we can print all the elements as shown in Figure 9.15. Here,



```

list_traversing.py - /home/kite/PythonWor...
File Edit Format Run Options Window Help
expenses = [120, 250, 80, 150, 200]
print("Expenses:")
for expense in expenses:
    print (expense)
Ln: 2 Col: 0

File Edit Shell Debug Options Window Help
Expenses:
120
250
80
150
200
Ln: 12 Col: 0

```

9.15: Traversing a list directly using a loop

```

count_score.py - /home/kite/PythonWorkbench/...
File Edit Format Run Options Window Help
marks = [55, 72, 38, 90, 25, 67, 48, 33, 17, 12, 60]
count = 0
for mark in marks:
    if mark < 40:
        count += 1
print('Number of students who scored below 40:', count)
Ln: 3 Col: 0

File Edit Shell Debug Options Window Help
Number of students who scored below 40: 5
Ln: 7 Col: 0

```

Figure 9.16: Count number of students with marks < 40

However, if we want to modify or update specific elements, we need to use index-based traversal, where we iterate through the list using indices. This approach is also called iteration by position.

9.7.2 Index-Based Traversal

```

traversing_marklist.py - /home/kite/PythonWorkbe...
File Edit Format Run Options Window Help
marks = [55, 72, 38, 90, 25, 17, 48, 33, 27, 12, 60]
pass mark = 25
for i in range(len(marks)):
    if marks[i] < pass mark:
        print("Sl.No.", i+1, "Failed")
Ln: 3 Col: 0

File Edit Shell Debug Options Window Help
Sl.No. 6 Failed
Sl.No. 10 Failed
Ln: 8 Col: 0

```

Figure 9.17: Traversing a Mark List Using Index-Based Access

each element from the expenses list is assigned to a variable named expense, which is then printed.

Program in Figure 9.16 illustrates a useful application of traversing a list. It counts how many students scored less than 40 marks from the list marks.

The direct traversal method works well when we only need to read or check each element of the list.

This method uses the `range()` function along with `len()` to generate index values. This idea has already been discussed in Chapter 6. By using the index, we have direct control over each element of the list and can perform specific actions based on conditions. See the program given in Figure 9.17, which iterates over a list and during the iteration it examines the elements for a particular case.

In this example, the `range(len(marks))` generates the indices from 0 to 9. The program checks each mark using its index and prints the serial number of students who have failed.

Traversal is not used just for displaying the elements of a list - we can also modify selected elements or perform operations on the entire list. It is often necessary to modify a list while traversing it, depending on the requirements of the problem. Consider the

```

list_processing.py - /home/kite/PythonWorkbench/List/list_processing.p...
File Edit Format Run Options Window Help
price = [200, 300, 75, 110, 300, 475, 500, 150, 600, 525]
print("Initial price list:", price)
for i in range(len(price)):
    if price[i] < 150:
        price[i] = price[i] + price[i] * 0.10
print("Final price list:", price)
Ln: 2 Col: 0

Initial price list: [200, 300, 75, 110, 300, 475, 500, 150, 600, 525]
Final price list: [200, 300, 82.5, 121.0, 300, 475, 500, 150, 600, 525]
>>>
Ln: 8 Col: 0

```

Figure 9.18: Traversing and processing a list

example shown in Figure 9.18, where the price of certain commodities is increased by 10% if the original price is below 150.

Now, let us modify the above program with following rules using a function.

If the price is below ₹200, it is increased by 10%.

- If the price is ₹500 or above, it is reduced by 5%.
- All other prices remain unchanged.

This program defines a function `update_price()` (Figure 9.19(a)) that receives a list of prices and modifies each value

using index-based traversal. Prices below ₹200 are increased by 10%, and prices equal to or above ₹500 are reduced by 5%. The updated list is then returned to the main program and displayed. This example shows how a function can make programs more organised and reusable. The output of the program is given in Figure 9.19 (b).

```

update_price.py - /home/kite/PythonWorkbench/List/update_...
File Edit Format Run Options Window Help

#Function to Update Price
def update_price(price_list):
    for i in range(len(price_list)):
        if price_list[i] < 200:
            price_list[i] = price_list[i] + price_list[i] * 0.10
        elif price_list[i] >= 500:
            price_list[i] = price_list[i] - price_list[i] * 0.05
    return price_list

#Main Program
price = [200, 300, 75, 110, 300, 475, 500, 150, 600, 525]
print("Initial Prices:", price)
new_price = update_price(price)
print("Updated Prices:", new_price)
Ln: 10 Col: 0
  
```

Figure 9.19 (a): Function performing calculations on list items

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help

Initial Prices: [200, 300, 75, 110, 300, 475, 500, 150, 600, 525]
Updated Prices: [200, 300, 82.5, 121.0, 300, 475, 475.0, 165.0, 570.0, 498.75]
Ln: 8 Col: 0
  
```

Figure 9.19(b): Output of list traversal program



Let us do

1. A Python program and its output are given in Figure 9.20 . Explain the output.
2. Consider the above program predict the output if `data = [5, 10, 15, 20]`. Also explain how the list changes during each iteration.
3. Write a Python program that contains the following:

A function `ReadList()` :- Input: No input, Return value: A list L.

Functional requirement: Create an empty list L, and read a sequence of inputs from the user and append the inputs into the list L in the given order, until the user inputs "None". The existing List L can be returned without appending "None" to the list. If the user enters only "None", then an empty list must be returned.

A function `PrintList(L)`:- Input: A list L, Return value: No return value.

Functional requirement: The contents of the L, if any, are displayed in the order in which they appear in the list. Note: The functions `ReadList()` and `PrintList()` can be easily modified to work even if L is a tuple, dictionary, set etc.

```

detail_output.py - /home/kite/Py...
File Edit Format Run Options Window Help

data = [5,10,15]
for i in range(len(data)):
    data[i] = data[i]*2
print(data)
Ln: 2 Col: 0

File Edit Shell Debug Options Window Help

[10, 10, 15]
[10, 20, 15]
[10, 20, 30]
Ln: 9 Col: 0
  
```

Figure 9.20: List modification using a for loop

The main code shall do the following. a) Invoke the function `ReadList()`. Assign the return value to a variable (say `L`). b) If `L` is not empty Invoke `PrintList()` to print the contents of `L`. Otherwise print "No data in the list".

Test your program by executing the code, and providing the names of all the municipal corporations of Kerala State as the input. Try other input combinations as well. The functions `ReadList()` and `PrintList()` will be useful in some later programming activities. Hence, you can put them into a module with the name *MyListModule.py*.

4. Write a Python program that contains the following:

The functions `ReadList()` and `PrintList()` defined in the previous question will be included.

The main code shall do the following: a) Invoke the function `ReadList()`. Assign the return value to a list `L`. b) Invoke `PrintList()` to print the contents of the list `L`. Do the following steps in a loop. # Recall loops earlier.

Ask the user to enter an index and read the value entered to a variable `k`.

If `k` is not a positive integer, then exit the loop.

Ask the user to enter data, and read the value entered to a variable.

If `k` $\leq$ number of elements in the list, then replace the `k`-th element of the list `L` with `s`. If `k` $>$ number of elements in the list, then append the string `s` to the list `L`. Print the new contents of `L` using the `PrintList()` function. Continue to the beginning of the loop. After the loop exits, print "Thank you" and exit the program.

Test your program by running it and giving "apple", "mango", "banana" as input initially. Next, replace "mango" with "jackfruit". Finally, you can try adding "orange" to the end of the list before exiting the program. You must try more examples yourself and ensure that your program works according to the given specification under all input conditions.

5. Write a Python program that contains the functions `ReadList()` and `PrintList()` from the previous question. In this question, our requirement is the following:
- Read a list of numbers from the user using `ReadList()`. Count the number of odd and even numbers in the list, and print the respective count values. For this purpose, create a function `OddList()` that takes as argument a list `L` and returns the number of odd integer values in the list. If `L` is empty, or does not contain any odd integer, the function should return `None`. Note that once we have counted odd numbers, then the count of even numbers can be easily computed.
 - Print "The smallest number in the list is:" followed by the smallest value in the list. For this purpose, you must create a function `FindSmallest()` that takes as argument a list `L` and returns the smallest number in `L`. If the list fails to contain any integer/float value, then the function must return `None`.

Test your program by entering the numbers 3, 7, 8, 9, 15. You must test your program with various combinations and check whether the program works correctly in all test cases. Remember that Python has a built-in function which allows you to check whether an element in a list is of integer type.

9.8 Applications of List

Lists provide convenient ways to store, manage, and manipulate data in Python. Their flexibility and dynamic nature make them suitable for a wide range of applications. In this section, we

will get familiar with some of the vital applications of lists in programming. Important algorithms such as linear search can be easily demonstrated using lists. Lists also support nested structures, making it possible to represent matrices and perform operations like addition and accessing diagonal elements. They can even help illustrate recursion, such as reversing a list, showing how lists support both simple and advanced computational tasks.

9.8.1 Linear Search

There are many real-world applications of lists in Python, and one of the most common among them is searching. Searching means finding whether a particular element exists within a collection of data, such as a list, array, or database, and if it does, determining its position. One of the simplest searching techniques is Linear Search. It is also known as sequential search, where elements in the list are checked one by one in order, starting from the first element, until the target element is found or the end of the list is reached. Although Python allows us to check membership easily using the `in` and `not in` operators, it is important to understand how searching happens step by step inside a list. Linear search helps us learn the procedure behind locating an element manually, without using built-in shortcuts.

Assume that we have a list of integer elements comprising 50, 18, 48, 35, 45, 26, 12. The element 45 is to be searched from this sequence. Linear search starts from the first element 50, comparing each element until it reaches the 5th position where it finds 45 as shown in Figure 9.21(a). Based on this, we can formulate the algorithm for linear search as given below.

Index	List	Comparison
0	50	50 == 45 : False
1	18	18 == 45 : False
2	48	48 == 45 : False
3	35	35 == 45 : False
4	45	45 == 45 : True
5	26	
6	12	

Figure 9.21(a): Linear search compares the target with each element.

Step 1: Start.

Step 2: Accept a value in `N` as the number of elements of the list.

Step 3: Accept `N` elements into the list `LIST`.

Step 4: Accept the value to be searched in the variable `ITEM`.

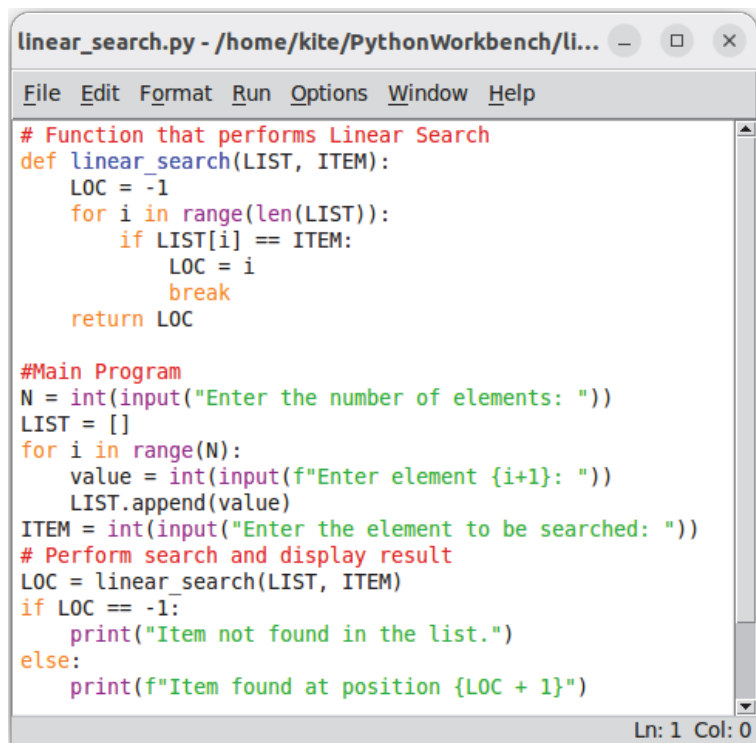
Step 5: Set `LOC = -1`.

Step 6: Starting from the first position, repeat Step 7 until the last element.

Step 7: Check whether the value in `ITEM` is found in the current position. If found then store the position in `LOC` and Go to Step 8, else move to the next position.

Step 8: If the value of `LOC` is less than 0 then display "Not Found", else display the value of `LOC + 1` as the position of the search value.

Step 9: Stop.



```
linear_search.py - /home/kite/PythonWorkbench/li...
File Edit Format Run Options Window Help

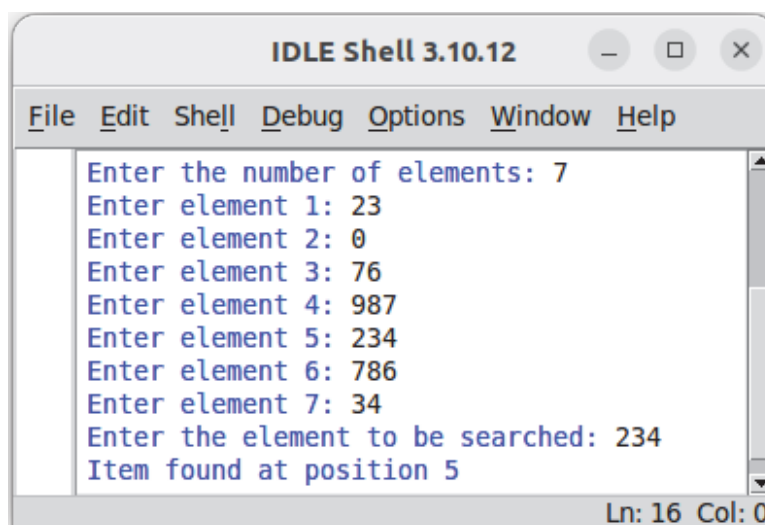
# Function that performs Linear Search
def linear_search(LIST, ITEM):
    LOC = -1
    for i in range(len(LIST)):
        if LIST[i] == ITEM:
            LOC = i
            break
    return LOC

#Main Program
N = int(input("Enter the number of elements: "))
LIST = []
for i in range(N):
    value = int(input(f"Enter element {i+1}: "))
    LIST.append(value)
ITEM = int(input("Enter the element to be searched: "))
# Perform search and display result
LOC = linear_search(LIST, ITEM)
if LOC == -1:
    print("Item not found in the list.")
else:
    print(f"Item found at position {LOC + 1}")

Ln: 1 Col: 0
```

Figure 9.21(b): Program that performs linear search

Now let us look at the Python implementation of the linear search algorithm. In the program shown in Figure 9.21(b), a function named `linear_search()` performs the search operation. The function accepts two arguments: the list containing the elements (LIST) and the item to be searched (ITEM). It checks whether each element is in order and returns the position of the item if it is found. After defining the function, it is invoked in the main program to carry out the search. Output of the program is given in Figure 9.21 (c).



```
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help

Enter the number of elements: 7
Enter element 1: 23
Enter element 2: 0
Enter element 3: 76
Enter element 4: 987
Enter element 5: 234
Enter element 6: 786
Enter element 7: 34
Enter the element to be searched: 234
Item found at position 5

Ln: 16 Col: 0
```

Figure 9.21 (c): Output of the linear search



Let us do

Based on the algorithm of linear search, prepare a flow chart for the same.

As pointed out earlier, Python provides many built-in functions to work with lists. However, it is both useful and interesting to understand how some of these operations are actually implemented within a program. By learning the internal logic behind these operations, we gain deeper insight into how programming languages handle data and can build our own algorithms when needed. With this intention, let us now explore an important mathematical concept and its implementation using lists in Python.

9.8.2 Matrix Representation

The idea of grouping related elements inside lists becomes even more powerful when we use nested lists to represent structured data. In data science, a matrix is an important concept that allows us to arrange numbers or information in a structured, two-dimensional format. A matrix is essentially a rectangular arrangement of values organised in rows and columns. Lists provide a convenient way to represent matrices through nested lists. This nested list structure helps us organise data in a tabular form, similar to the rows and columns of a table.

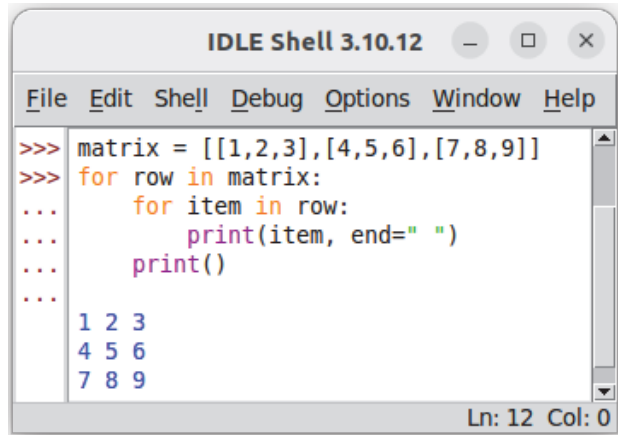
Using inner lists, we can represent a matrix. When all the inner lists contain the same number of elements, the nested list forms a valid matrix. While creating such a structure, each inner list corresponds to a **row** of the matrix, and the elements aligned vertically across these lists together form the **columns** of the matrix. Various operations can be performed on matrices, such as displaying elements, accessing specific values, and performing arithmetic operations like addition. In a nested list, each inner list represents a row, and the number of elements in each row represents the columns. A matrix is usually denoted in the order *row × column* ($m \times n$), where m is the number of rows and n is the number of columns. For example, a 3×3 matrix can be represented as:

$$matrix = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

The program in Figure 9.22 shows how a matrix can be created and printed using lists. Note that this program is written in interactive mode. Here, we use nested lists to represent the matrix and two **for** loops to display it. The outer loop iterates through each inner list, which represents a row of the matrix, and the inner loop goes through each element within that row. The `print(item, end=" ")` statement prints the elements of a row on the same line, while the `print()` placed after the inner loop moves the cursor to the next line, ensuring that each row appears separately just like a matrix.

Accessing Elements of a Matrix

Each element of a matrix can be accessed using its address value, which specifies its row and column. The general form is: `matrix[row][column]`



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> matrix = [[1,2,3],[4,5,6],[7,8,9]]
>>> for row in matrix:
...     for item in row:
...         print(item, end=" ")
...     print()
...
1 2 3
4 5 6
7 8 9
Ln: 12 Col: 0

```

Figure 9.22: Matrix representation using nested lists

- The row number comes first, followed by the column number.
- Row and column numbering usually starts from 0 in programming languages like Python.

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

For example

- `matrix[0][0]` refers to the element in Row 1, Column 1, which is 1.
- `matrix[1][2]` refers to the element in Row 2, Column 3, which is 6.
- `matrix[2][1]` refers to the element in Row 3, Column 2, which is 8.

We can print individual elements directly using the `print()` statement as shown below:

```
print(matrix[0][2]) # First row, third column
print(matrix[2][1]) # Third row, second column
```

Accepting Matrix

Now let us create a matrix by accepting values from the console. The program shown in Figure 9.23(a) demonstrates how to accept the

elements of a matrix from the user and display the matrix in proper tabular form. The program uses two separate functions one for accepting values and another for printing the matrix. Using functions makes the program more organised, reusable, and easier to understand.

The `accept_matrix()` function first asks the user to enter the number of rows

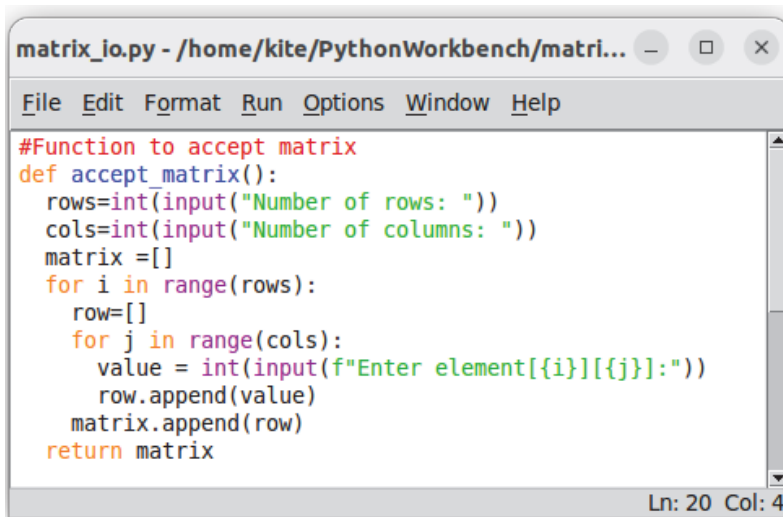


Figure 9.23 (a): Function for reading a matrix from user input

and columns for the matrix. An empty list named `matrix` is then created to store all the rows. The statement

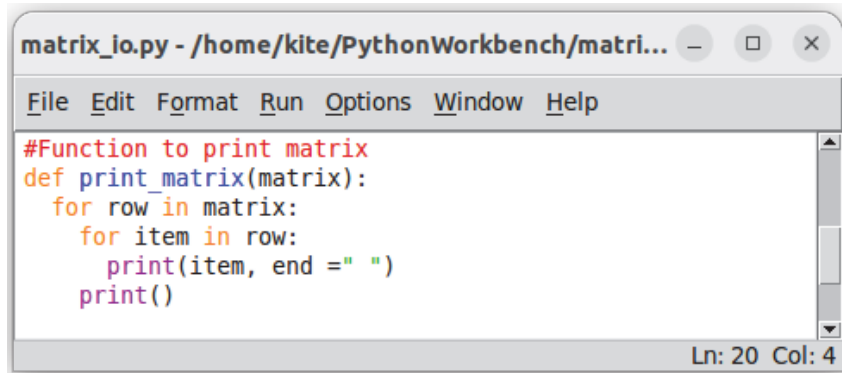
`input(f"Enter element [{i}][{j}]: ")` provides a clear and user-friendly prompt for entering each element. To build the matrix, two nested for loops are used:

- Outer loop (`for i in range(rows)`): Creates each row of the matrix.
- Inner loop (`for j in range(cols)`): Accepts each element of that row.

For every element position, the user is prompted to enter a value, which is then appended to the list named `row`. Once all the elements

of a row are entered, that row list is appended to the main list `matrix`. This process continues until all rows are completed. After the matrix is fully constructed, the function returns the completed matrix to the calling program. Thus, this function enables the dynamic creation of a matrix of any size specified by the user.

The function `print_matrix(matrix)` (shown in Figure 9.23(b)) is used to display the elements of a matrix in a structured row-column format. It receives a matrix (a list of lists) as an argument. The function uses a nested loop to access and print each element of the matrix.



```
matrix_io.py - /home/kite/PythonWorkbench/matri...
File Edit Format Run Options Window Help

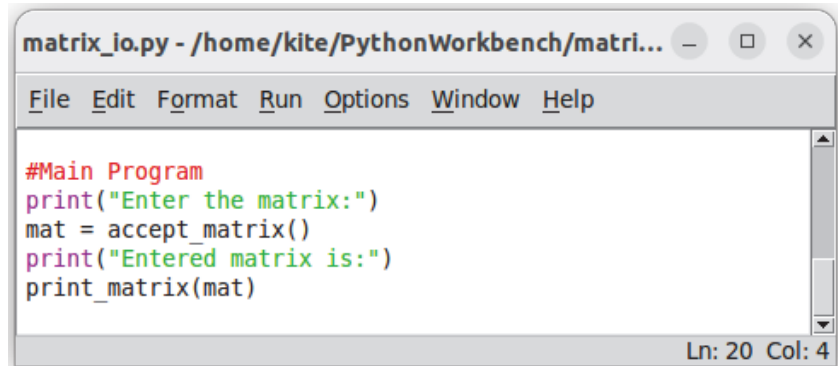
#Function to print matrix
def print_matrix(matrix):
    for row in matrix:
        for item in row:
            print(item, end=" ")
        print()

Ln: 20 Col: 4
```

Figure 9.23(b): Function to print matrix

The outer loop (`for row in matrix:`) iterates through each row of the matrix. Since each row itself is a list, the inner loop (`for item in row:`) processes every element within that row. The statement `print(item, end=" ")` prints each element on the same line, separated by spaces. The parameter `end=" "` ensures that the output remains on one line until the row is complete.

After all items in a row are printed, the statement `print()` moves the cursor to the next line, ensuring that the next row appears on a new line. Thus, the function displays the matrix exactly as in standard mathematical form.



```
matrix_io.py - /home/kite/PythonWorkbench/matri...
File Edit Format Run Options Window Help

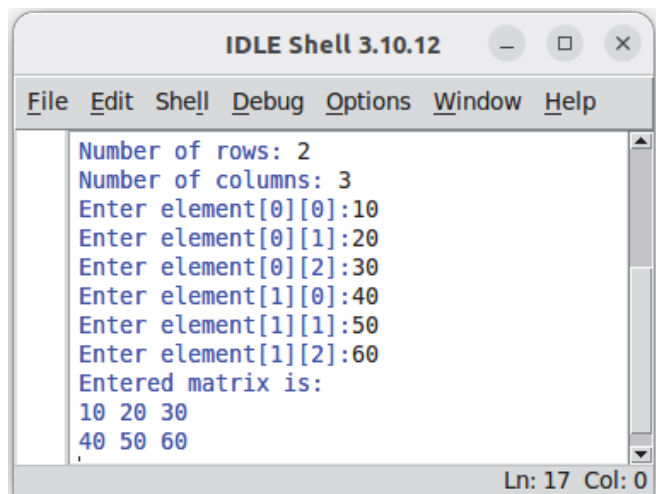
#Main Program
print("Enter the matrix:")
mat = accept_matrix()
print("Entered matrix is:")
print_matrix(mat)

Ln: 20 Col: 4
```

Figure 9.23 (c): Main program that invokes the required functions

In the main program (Shown in Figure 9.23(c)), the previously defined functions are invoked to perform the required tasks.

The output of the program is given in Figure 9.23(d). The step-by-step process of building the matrix through each loop iteration is shown in Table 9.2.



```
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help

Number of rows: 2
Number of columns: 3
Enter element[0][0]:10
Enter element[0][1]:20
Enter element[0][2]:30
Enter element[1][0]:40
Enter element[1][1]:50
Enter element[1][2]:60
Entered matrix is:
10 20 30
40 50 60

Ln: 17 Col: 0
```

Figure 9.23(d): Output of the matrix-printing program

Table 9.2: Sequence of operations while executing the loop

Step	Input Value	Value of row	Value of matrix
1	10	[10]	-
2	20	[10, 20]	-
3	30	[10, 20, 30]	[[10, 20, 30]]
4	40	[40]	[[10, 20, 30]]
5	50	[40, 50]	[[10, 20, 30]]
6	60	[40, 50, 60]	[[10, 20, 30], [40, 50, 60]]

See Figure 9.23 (e) for more clarity. It shows how each row and the entire matrix are updated during different iterations of the loops. This illustration is based on the output screen shown in Figure 9.23(d)

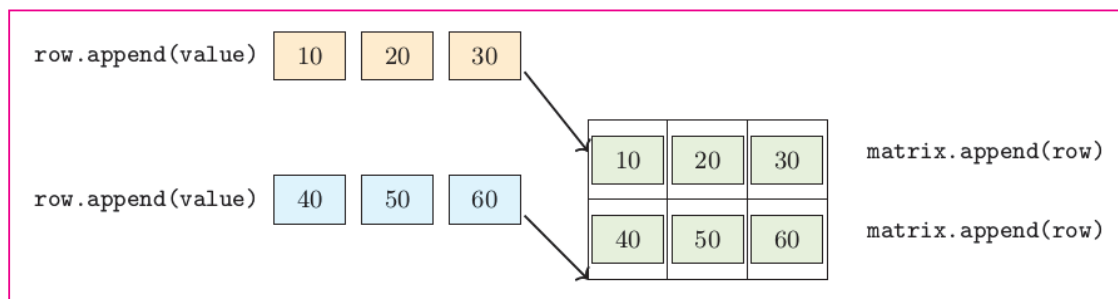


Figure 9.23(e): Matrix creation in Python using append()

9.8.3 Matrix Addition

Similar to how we add numbers, we can also add two matrices. However, before performing matrix addition, it is important to understand the **order** of a matrix. Two matrices can be added only if they **have the same order** (i.e., the same number of rows and columns). Only then are they considered conformable for addition.

When adding matrices, we simply add the corresponding elements (same row and same column positions) to form a new matrix.

```

matrix_adder.py - /home/kite/PythonWorkb...
File Edit Format Run Options Window Help
# Function to add two matrices
def add_matrices(mat1, mat2):
    rows = len(mat1)
    cols = len(mat1[0])
    # Create an empty matrix to store the result
    result = []
    for i in range(rows):
        row = []
        for j in range(cols):
            # Add corresponding elements
            row.append(mat1[i][j] + mat2[i][j])
        result.append(row)
    return result
Ln: 1 Col: 0

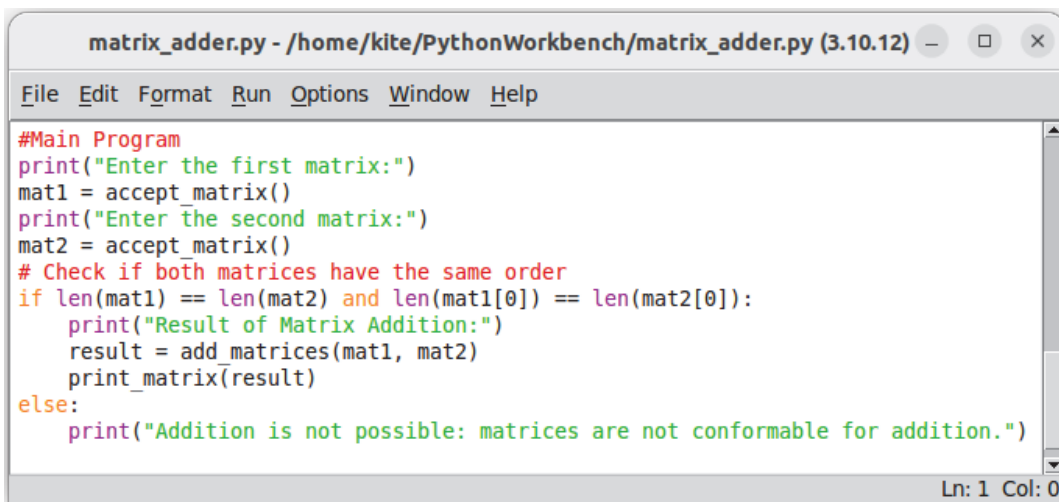
```

Figure 9.24 (a): Function that performs matrix addition

The function `add_matrices (mat1, mat2)` shown in Figure 9.24(a) performs matrix addition by adding corresponding elements of two matrices. First, it determines the number of rows and columns using `len()`. Then, an empty list named `result` is created to store the final matrix. Using nested loops, the outer loop iterates through each row, and the inner loop accesses each column element.

For every position $[i][j]$, the function adds the element from the first matrix $mat1[i][j]$ to the element from the second matrix $mat2[i][j]$ and stores the sum in a temporary list named `row`. After completing each row, it is appended to the result matrix. Once all rows and columns are processed, the function returns the resultant matrix, which contains the element-wise sum of the two input matrices.

Now let us see how all the functions are invoked together in the main program. The program shown in Figure 9.24(b) manages matrix addition operation. The function `accept_matrix()` is reused twice to create two separate matrices, `mat1` and `mat2`.



```

matrix_adder.py - /home/kite/PythonWorkbench/matrix_adder.py (3.10.12)
File Edit Format Run Options Window Help

#Main Program
print("Enter the first matrix:")
mat1 = accept_matrix()
print("Enter the second matrix:")
mat2 = accept_matrix()
# Check if both matrices have the same order
if len(mat1) == len(mat2) and len(mat1[0]) == len(mat2[0]):
    print("Result of Matrix Addition:")
    result = add_matrices(mat1, mat2)
    print_matrix(result)
else:
    print("Addition is not possible: matrices are not conformable for addition.")
Ln: 1 Col: 0

```

Figure 9.24(b): Main Program that manages matrix addition

The function `accept_matrix()` is already discussed in previous section, see Figure 9.23(a). This demonstrates reusability, since we use the same function instead of writing repeated code. Before performing addition, the program checks whether the matrices are compatible for addition by comparing their orders, both matrices must have the same number of rows and columns.

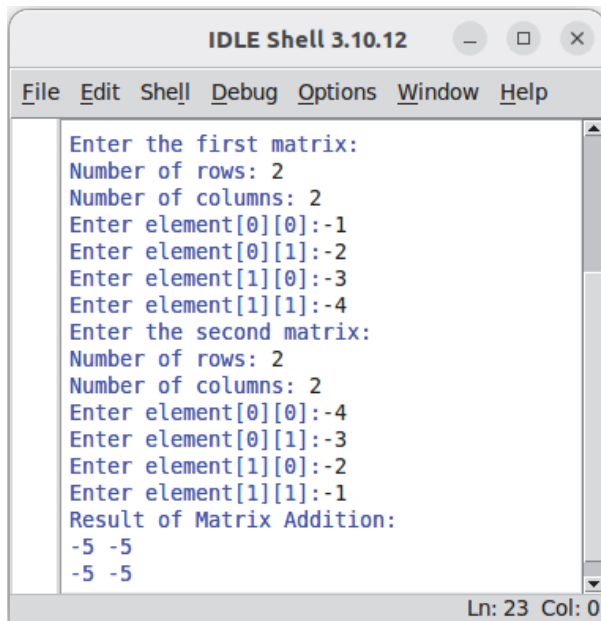
Validating the order of matrix

In the main program (shown in Figure 9.24(b)) the following if condition is used to verify whether matrix addition is possible or not.

```
if len(mat1) == len(mat2) and len(mat1[0]) == len(mat2[0]):
```

- `len(mat1)` → number of rows in the first matrix
- `len(mat2)` → number of rows in the second matrix
- `len(mat1[0])` → number of columns in the first matrix
- `len(mat2[0])` → number of columns in the second matrix

If the orders match, the function `add_matrices(mat1, mat2)` is invoked to compute element-wise addition, and the resultant matrix is displayed using the `print_matrix()` function. Thus, matrix creation, validation, addition, and printing are managed smoothly



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Enter the first matrix:
Number of rows: 2
Number of columns: 2
Enter element[0][0]:-1
Enter element[0][1]:-2
Enter element[1][0]:-3
Enter element[1][1]:-4
Enter the second matrix:
Number of rows: 2
Number of columns: 2
Enter element[0][0]:-4
Enter element[0][1]:-3
Enter element[1][0]:-2
Enter element[1][1]:-1
Result of Matrix Addition:
-5 -5
-5 -5
Ln: 23 Col: 0

```

Figure 9.24(c): Matrix addition output

through sequential function calls in the main program. Figure 9.24(c) gives the output of the program.

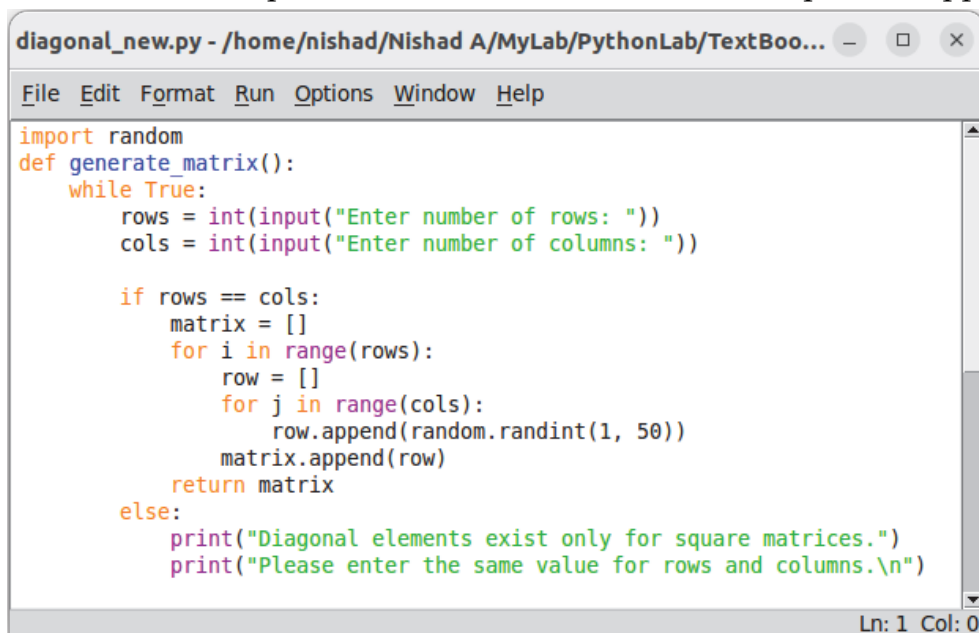
9.8.4 Processing Diagonal Elements of a Matrix

Diagonal elements of a matrix are the elements that lie from the top-left corner to the bottom-right corner. These elements occur at positions where the row index and the column index are the same, that is, at $[i][i]$, as given below.

$$\begin{bmatrix}
 a_{11} & a_{12} & a_{13} \\
 a_{21} & a_{22} & a_{23} \\
 a_{31} & a_{32} & a_{33}
 \end{bmatrix}$$

Diagonal elements are important in many matrix operations such as finding the trace, simplifying calculations, and identifying special types of matrices. A program to print the diagonal elements of a square matrix using functions is illustrated here. Instead of accepting matrix elements from the user, this program automatically generates matrix values using a random number generator.

The function `generate_matrix()` shown in Figure 9.25(a) is responsible for creating the matrix. The user is asked to enter the number of rows and columns. Since diagonal elements exist only in square matrices, the function first validates whether the entered order is square, that is, whether the number of rows and columns are equal. If the entered values are not equal, an appropriate message



```

diagonal_new.py - /home/nishad/Nishad A/MyLab/PythonLab/TextBoo...
File Edit Format Run Options Window Help
import random
def generate_matrix():
    while True:
        rows = int(input("Enter number of rows: "))
        cols = int(input("Enter number of columns: "))

        if rows == cols:
            matrix = []
            for i in range(rows):
                row = []
                for j in range(cols):
                    row.append(random.randint(1, 50))
                matrix.append(row)
            return matrix
        else:
            print("Diagonal elements exist only for square matrices.")
            print("Please enter the same value for rows and columns.\n")
Ln: 1 Col: 0

```

Figure 9.25(a): Program that generates square matrix

is displayed and the user is asked to re-enter the values. This validation is performed using a **while** loop, which continues to execute until the user provides equal values for rows and columns, ensuring that only a valid

square matrix is created before proceeding further. If the entered values are not equal, the **else** part of the **while** loop displays an error message and prompts the user to enter the values again.

Once a valid square order is provided, the function generates a matrix filled with random integers using the **randint()** function from the **random** module. This module is useful for generating random numbers and is especially helpful when testing programs with sample data or simulating values automatically. The randomly generated values are stored row-wise inside a list, forming the complete matrix. Here, integer values in the range 1 to 50 are generated using the instruction `random.randint(1, 50)`.

Before printing the diagonal elements, the function `print_matrix(matrix)` already discussed is invoked to display the generated matrix in proper matrix format. The function `print_diagonal(matrix)` (Figure 9.25(b)) prints only the diagonal elements by accessing the positions `[i][i]` using a single loop. This method works correctly because the matrix is ensured to be square through validation.

The main program (shown in Figure 9.25(c)) controls the execution by invoking the functions `generate_matrix()`, `print_matrix` (`matrix`), and `print_diagonal(matrix)`.

The output of the program is given in Figure 9.25(d).

It's Random, But Not Accidental: The Logic Behind Chance



Have you ever wondered how computer games create unpredictable moments, like the roll of a dice or the random appearance of a collectible? This is managed through the concept of **randomness**, which is simulated in computers using a Pseudo-Random Number Generator (PRNG). Since computers are deterministic machines that follow strict instructions, they cannot produce truly random numbers without special hardware. Instead, a PRNG uses a mathematical algorithm to generate a long sequence of numbers that appear random and are unpredictable for most practical purposes. In this sequence, each number in a given range has an equal probability of being selected. In Python, this power is readily available in the **random** module, allowing you to use functions like `randint()` to simulate a dice roll. This ability to generate randomness is not just mathematically interesting; it is a cornerstone of countless real-world applications, from games and simulations to cryptography and scientific modelling.

```
diagonal_new.py - /home/nishad/Nish...
File Edit Format Run Options Window Help
def print_diagonal(matrix):
    print("Diagonal Elements:")
    for i in range(len(matrix)):
        print(matrix[i][i], end=" ")
Ln: 1 Col: 0
```

Figure 9.25(b): Print diagonal elements of a matrix.

```
diagonal_new.py - /home/nishad/Nish...
File Edit Format Run Options Window Help
# Main Program
matrix = generate_matrix()
print("Generated Matrix:")
print_matrix(matrix)
print_diagonal(matrix)
Ln: 1 Col: 0
```

Figure 9.25(c): Main program that manages various functions

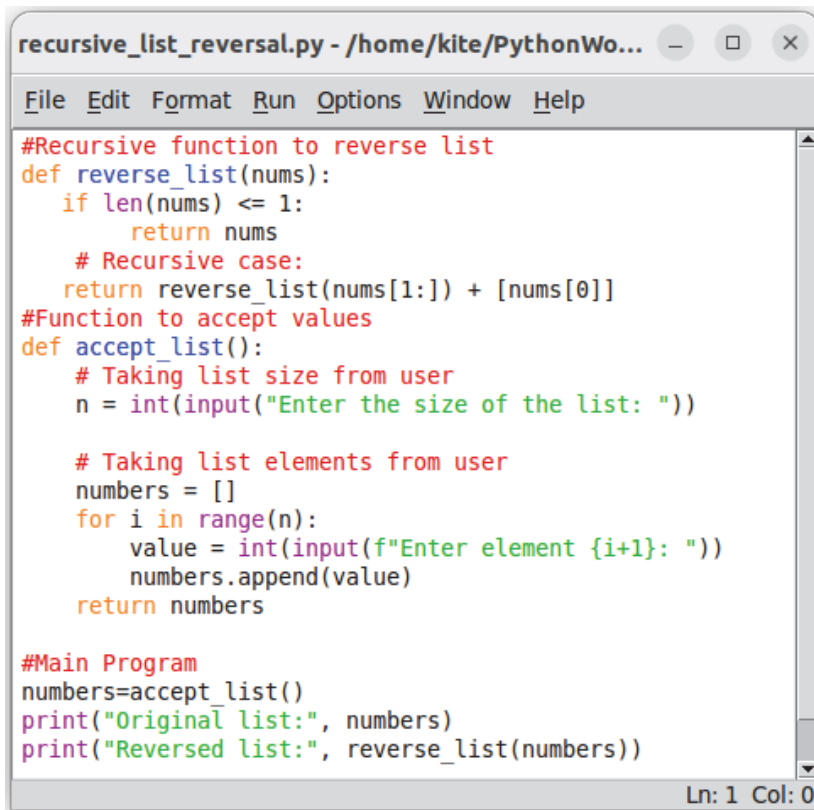
```
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Enter number of rows: 4
Enter number of columns: 3
Diagonal elements exist only for square matrices.
Please enter the same value for rows and columns.

Enter number of rows: 3
Enter number of columns: 3
Generated Matrix:
2    42    1
49   44   38
17   34   22
Diagonal Elements:
2 44 22
Ln: 19 Col: 0
```

Figure 9.25(d): Output - Diagonal Elements of a matrix

9.8.5 Reverse a List Using Recursion

We have already learned the concept of recursion in Chapter 7. Now, let us develop an application with list. See the program shown in Figure 9.26(a).



```

recursive_list_reversal.py - /home/kite/PythonWo...
File Edit Format Run Options Window Help

#Recursive function to reverse list
def reverse_list(nums):
    if len(nums) <= 1:
        return nums
    # Recursive case:
    return reverse_list(nums[1:]) + [nums[0]]
#Function to accept values
def accept_list():
    # Taking list size from user
    n = int(input("Enter the size of the list: "))

    # Taking list elements from user
    numbers = []
    for i in range(n):
        value = int(input(f"Enter element {i+1}: "))
        numbers.append(value)
    return numbers

#Main Program
numbers=accept_list()
print("Original list:", numbers)
print("Reversed list:", reverse_list(numbers))
Ln: 1 Col: 0
  
```

Figure 9.26(a): Recursive program to reverse a list

the first element and then appends the first element to the end, gradually building the reversed list.

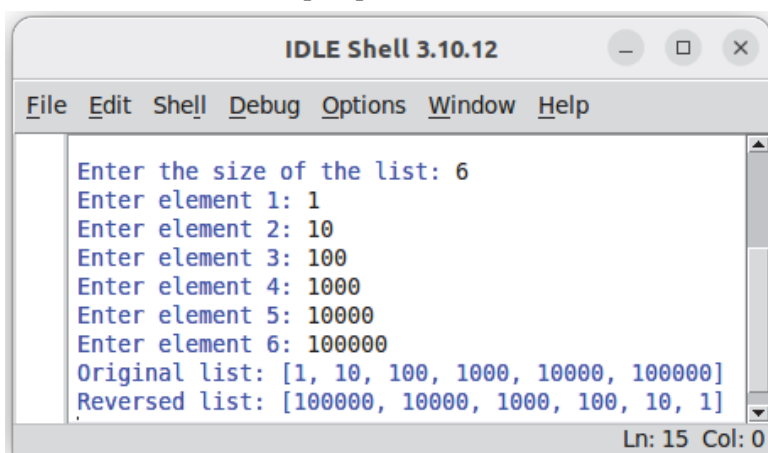
The program first asks the user to enter the size of the list and then uses a loop to input each element, storing them in the list numbers. Finally, it prints both the original list and the reversed list to show the result of the recursive process. The statement

```
return reverse_list(nums[1:]) + [nums[0]]
```

forms the core logic of the recursive list reversal. The expression `nums[1:]` creates a new list that excludes the first element of `nums`.

The function then calls itself with this shortened list:

- `reverse_list(nums[1:])` returns the reversed form of all elements except the first.
- The element `nums[0]`, which was removed in the slicing, is placed at the end of the reversed list by writing `[nums[0]]`.



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help

Enter the size of the list: 6
Enter element 1: 1
Enter element 2: 10
Enter element 3: 100
Enter element 4: 1000
Enter element 5: 10000
Enter element 6: 100000
Original list: [1, 10, 100, 1000, 10000, 100000]
Reversed list: [100000, 10000, 1000, 100, 10, 1]
Ln: 15 Col: 0
  
```

Figure 9.26(b): Output: Reversing a list using recursion

By combining these parts using the + operator, the function constructs the reversed list step by step. The output of the program is shown in in Figure 9.26(b).



Let us do

1. Write a Python program to analyse a set of user-provided numbers. Structure your code using functional decomposition by creating separate functions for: (a) collecting a list of n numbers from the user, (b) calculating the arithmetic average of the list, and (c) identifying and displaying only those numbers from the original list that are strictly greater than the calculated average.
2. Write a Python program to calculate the average value of all elements in a user-provided matrix, and then display all elements that are greater than that average. Structure your code using functional decomposition by creating separate functions for: (a) collecting the $N \times M$ matrix input from the user, (b) a function named `calculate_matrix_average()` that accepts the matrix and returns the floating-point average of all its elements, and (c) a function named `display_elements_above_average()` that accepts the matrix and the average, and then prints all elements strictly exceeding that average.
3. Write a Python program to determine the transpose of a matrix (swapping rows with columns). Structure the code using functional decomposition by creating separate functions for: (a) capturing user input, (b) computing the transpose logic, and (c) displaying the matrix in a standard row-column format.
4. Define the concept of a symmetric matrix ($A=A^T$). Write a Python program to check if a user-input square matrix is symmetric. Structure the solution with separate functions for: (a) collecting the square matrix input, (b) calculating the transpose, and (c) using a function named `is_symmetric()` to compare A and A^T and return a boolean result.
5. Write a Python program to find the trace of a square matrix. The trace of a matrix, often denoted $\text{tr}(A)$, is defined as the sum of the elements on its main diagonal. Structure your code using functional decomposition by creating separate functions for: (a) capturing the square matrix input from the user, and (b) a function named `calculate_trace()` that accepts the matrix and returns the integer sum of its diagonal elements.
6. Write a Python program to demonstrate matrix-scalar addition and subtraction. Structure your solution using functional decomposition by creating separate functions for: (a) generating a random $N \times M$ matrix of integers (with values between 0 and 100), (b) collecting a single integer scalar value from the user, (c) performing the element-wise matrix-scalar addition, (d) performing the element-wise matrix-scalar subtraction, and (e) displaying the resulting matrices.

In this chapter, we explored lists, one of Python's most versatile and widely used data structures. We learned different ways to create lists, how to access elements using indexing, and how to extract portions of a list through slicing. The chapter also demonstrated how lists support practical operations such as searching, modifying values, and representing structured data through nested lists. Applications like linear search and matrix operations highlighted the real-world usefulness of lists. With their flexibility, mutability, and ease of manipulation, lists form a strong foundation for solving a wide range of programming problems in Python. In the next chapter, we will learn about tuples—another important sequence type that differs from lists in significant ways.

Tuples

In Python, a tuple is an ordered collection of elements. Its key feature is immutability, which means that once created, its values cannot be changed, added, or removed. Because of this, tuples are often used to store fixed sets of information, for example geographical coordinates, dates, or any data that should remain unchanged. It is very similar to a list, except for its immutability.

In this chapter, we will learn how to work with tuples through various operations such as accessing elements, using positive and negative indexing, slicing with start-stop-step values, and performing operations like concatenation and membership tests. We will also understand why tuple immutability is important and how Python handles tuple objects in memory.

Since tuples cannot be changed, we will also explore how to convert between lists and tuples, how tuples can contain mutable elements such as lists, and how tuple assignment allows multi-variable swapping. By the end of this chapter, we will understand the role of tuples, how they differ from lists, and where each is best used in real-world programming and data handling.

10.1 Creating Tuple

A tuple is created by enclosing its elements within parentheses (), with each element separated by a comma. For example

```
>>>numbers = (10, 20, 30, 40)
>>>numbers
(10, 20, 30, 40)
```

We can create a tuple even without using parentheses. Python automatically treats a sequence of values separated by commas as a tuple. This method of creating a tuple by simply listing values separated by commas is called tuple packing. For example, consider the following tuple containing string elements:

```
>>>hilly_areas = "Munnar", "Ponmudi",
    "Sholayoor", "Vagamom"
>>>print(hilly_areas)
('Munnar', 'Ponmudi', 'Sholayoor', 'Vagamom')
```

Tuples do not have all the methods that lists have, for example, the `append()` method does not work with tuples because they are immutable.

Tuple supports holding mixed data types:

```
>>>t = ("apple","grapes", 2, 23, 2.5)
>>>print(t)
('apple', 'grapes', 2, 23, 2.5)
```

In order to create a tuple with a single element, we must include a comma. A single element inside parentheses without a comma is not considered a tuple. Without the comma, Python treats the value as a normal integer (or whatever type the value represents). We can verify this by checking the type of the variables using the `type()` function.

```
>>>t1 = (1984,)
>>>print(t1)
(1984,)
>>>type(t1)
<class 'tuple'>
>>>t2 = (78)
>>>type(t2)
<class 'int'>
```

An empty tuple can be created using just a pair of parentheses as given below.

```
>>>t3 = ()
```



Let us do

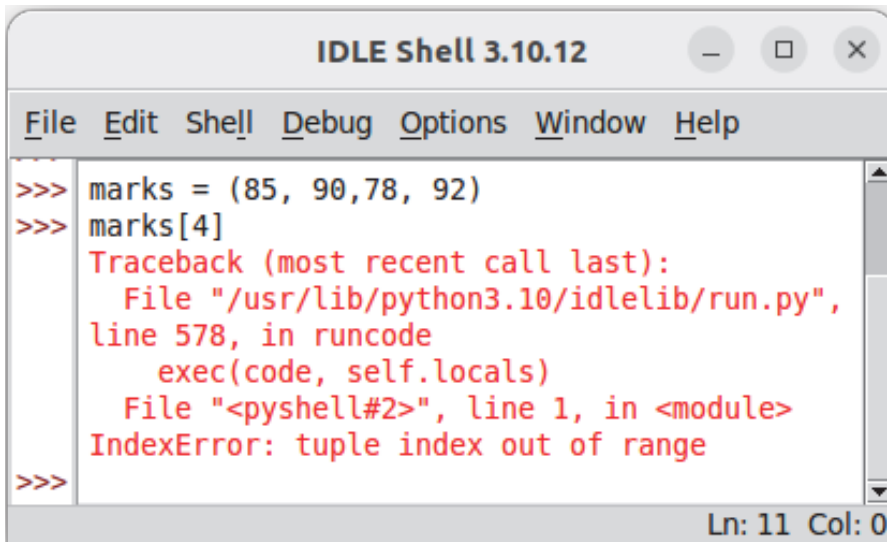
Write a Python program to create two variables: one using a single string inside parentheses without a comma, and another using the same string inside parentheses with a comma. Use the `type()` function to display the data type of each variable.

10.2 Accessing Tuple Elements

Every element in a tuple has a specific position called its index. In Python, indexing always begins at 0 for the first element. Using these index values, we can access any specific element in the tuple. Indexing in tuples is similar to indexing in lists.

```
>>>marks = (85, 90, 78, 92)
>>>print(marks[0])
85
>>>print(marks[3])
92
```


If we try to access an index that does not exist, Python will raise an error. See the example given in Figure 10.1, we attempt to access an element beyond the valid index range. Since the last valid index of the tuple `mark` is 3, trying to access index 4 results in an error.



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> marks = (85, 90, 78, 92)
>>> marks[4]
Traceback (most recent call last):
  File "/usr/lib/python3.10/idlelib/run.py", line 578, in runcode
    exec(code, self.locals)
  File "<pyshell#2>", line 1, in <module>
IndexError: tuple index out of range
>>>
Ln: 11 Col: 0

```

Figure 10.1: Error when accessing a non-existent index

Negative Indexing

Tuple also allows indexing from the right end using negative values. The rules for negative indexing in tuples are the same as those for lists. Consider the following examples.

```

>>> days = ("Monday", "Tuesday", "Wednesday",
            "Thursday", "Friday", "Saturday", "Sunday")
>>> print(days[-1])
Sunday
>>> print(days[-3])
Friday

```

Negative indexing is very useful when we want to access elements from the end without calculating their exact positions.

10.3 Slicing a Tuple

Sometimes, more than one element is required from a tuple. In such cases, a specific portion of the tuple can be extracted using slicing. The general form of slicing a tuple is given below.

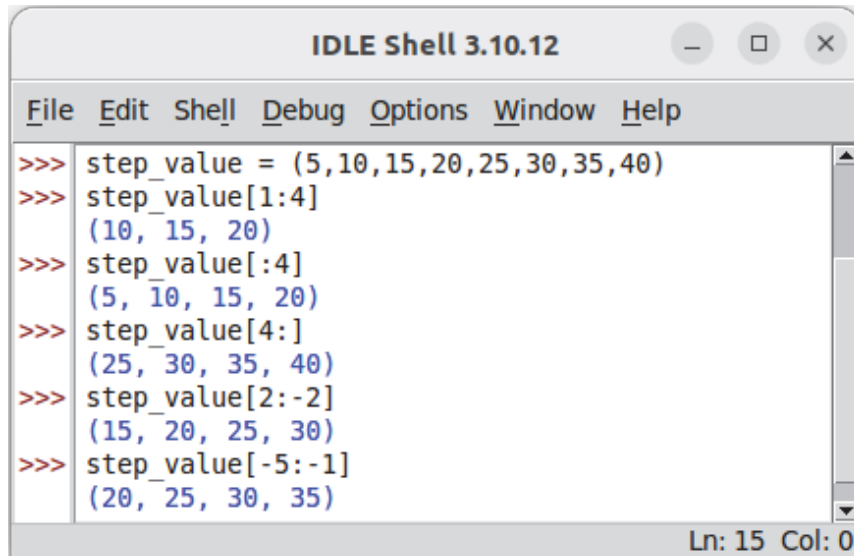
tuple_name[start : end]

`start` → index from where the slice begins

`end` → index where the slice stops (but not included)

If the `start` index in a slice is not specified, Python assumes it to be 0. Similarly, if the `end` index is not provided, the slice extends up to the end of the tuple. Negative indexing can also be used with tuple

slicing. Figure 10.2 shows various slicing options, all demonstrated using a tuple named `step_value`.



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> step_value = (5,10,15,20,25,30,35,40)
>>> step_value[1:4]
(10, 15, 20)
>>> step_value[:4]
(5, 10, 15, 20)
>>> step_value[4:]
(25, 30, 35, 40)
>>> step_value[2:-2]
(15, 20, 25, 30)
>>> step_value[-5:-1]
(20, 25, 30, 35)
Ln: 15 Col: 0

```

Figure 10.2: Examples of Tuple Slicing

When slicing a tuple, a third optional value called the step can also be specified. The step value determines how many positions Python skips after selecting each element. The rules for using a step in slicing are the same as those used for strings. The general form of tuple slicing with a step value is:

`tuple_name[start : end : step]`

For example, if the step value is '2', Python selects every second element. If the step value is '-1', the elements are accessed in reverse order. Table 10.1 illustrates various ways of slicing a tuple using different step values. Consider the tuple

`step_value = (5,10,15,20,25,30,35,40)`

Table 10.1 gives various slicing operations.

Table 10.1: Examples of Tuple Slicing with Different Step Values

Expression	Result	What it does
<code>step_value[::2]</code>	<code>(5, 15, 25, 35)</code>	Selects every second element from the beginning.
<code>step_value[1::2]</code>	<code>(10, 20, 30, 40)</code>	Starts from index 1 and takes every second element.
<code>step_value[2:7:3]</code>	<code>(15, 30)</code>	Starts at index 2, ends before index 7, steps by 3.
<code>step_value[::-1]</code>	<code>(40, 35, 30, 25, 20, 15, 10, 5)</code>	Reverses the entire tuple (step is negative).
<code>step_value[-2::-2]</code>	<code>(35, 25, 15, 5)</code>	Starts from second last index and goes back by 2 steps.



Let us do

- Given the tuple: `t = (12, 24, 36, 48, 60, 72)` write Python script to perform the following.
 - Print the first and third elements using positive indexing
 - Print the last element using negative indexing
 - Print the second last element using negative indexing
- Using the same tuple `t` write Python script to display:
 - First three elements
 - Elements from index 2 to the end
 - Three middle elements using slicing
- Predict the output and explain why:

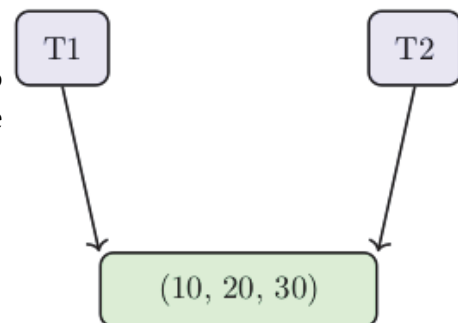

```
x = (10)
print(type(x))
y = (10,)
print(type(y))
```

10.4 Immutability of Tuple

Like any other literal collection, tuples are also objects stored in memory, and variable names are simply references (or labels) that point to these objects.

Consider two tuples T1 and T2

```
>>>T1 = (10, 20, 30)
>>>T2 = T1
```



Here, both T1 and T2 refer to the same tuple object in memory (See Figure 10.3), just as L1 and L2 referred to the same list earlier. However, since tuples are immutable, even though both variables point to the same object, we cannot modify the elements of that object.

When we attempt to overwrite an existing element in a tuple with a new value, Python raises an exception, as shown in Figure 10.4. This means we can only create a new tuple, but not change an existing one.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> T1 = (10,20,30)
>>> T1[0] = 99
Traceback (most recent call last):
  File "/usr/lib/python3.10/idlelib/run.py", line 578, in
runcode
    exec(code, self.locals)
  File "<pyshell#2>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> |
Ln: 11 Col: 0
  
```

Figure 10.4: Python Showing TypeError for Tuple Modification

10.5 Concatenation of Tuples

Concatenation means joining two or more tuples into a single tuple. This is done using the + operator.

```
>>>T1 = (10,20,30)
>>>T2 = T1+(40,)
>>>print(T2)
(10, 20, 30, 40)
```

In the above example, we are concatenating an integer tuple with another tuple containing a single element. At first glance, it may seem as if the tuple T1 has been modified by adding 40. However, this is not what actually happens. Python creates a new tuple in memory that contains all the elements of T1 along with the new element 40. The variable T2 now refers to this new tuple, while T1 still points to the original one. Figure 10.5 illustrates how the new tuple is created without altering the original. The program in

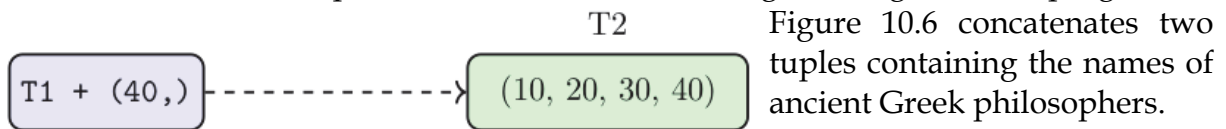


Figure 10.5: Concatenation Creates a New Tuple

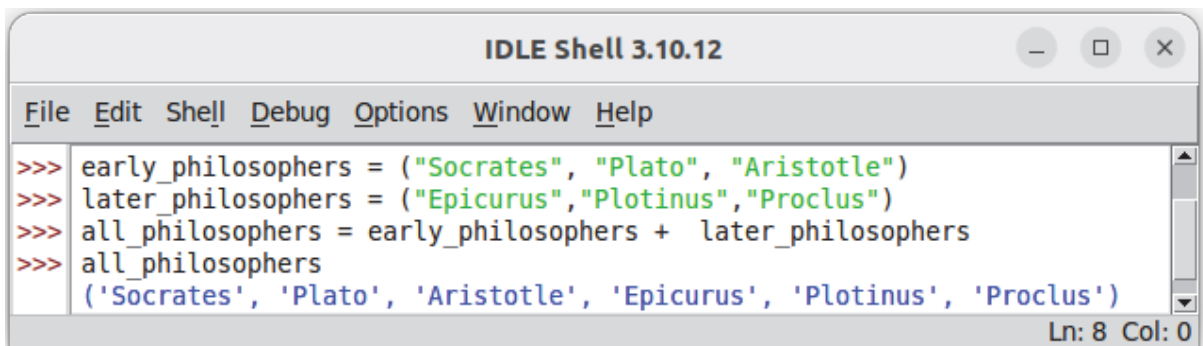


Figure 10.6: Concatenation of tuples with string elements

Tuple Unpacking

We learned that tuple packing is an alternative way of creating tuples. This feature is often used when a function returns multiple values. Tuple unpacking is the reverse of packing. It means taking the individual values stored inside a tuple and assigning them to separate variables in a single statement. Python automatically matches each element of the tuple to the corresponding variable on the left side

```
>>>t = (10, 20, 30)
>>>a, b, c = t
>>>print(a,b,c)
10 20 30
```

In the program, the statement `a, b, c = t` unpacks the tuple and the first value (10) is assigned to **a**, the second value (20) is assigned to **b**, the third value (30) is assigned to **c**. Tuple packing and unpacking are particularly useful in functions for handling variable-length arguments and returning multiple results.



Let us do

1. Write Python code to pack three details of a book (title, author, price) into a tuple and print it.
2. Write Python code that creates a tuple containing your name, class, and favourite subject. Print each element using indexing.
3. Write Python code that creates two tuples containing any sequence of numbers (for example, one tuple of odd numbers and another of even numbers). Concatenate the two tuples to form a single combined tuple, and print the final result.

10.6 Tuple Functions and Operations

Although tuples have very few built-in functions of their own, many of Python's general built-in functions can still be used to perform meaningful operations on them. In this section, we will explore some useful operations related to tuples.

10.6.1 Membership Operators

Like lists, tuples also support the membership operators **in** and **not in**. The operator **in** returns True if the element exists in the tuple, and **not in** returns True if the element does not exist in the tuple. See some examples.

```
>>>numbers = (10, 20, 30, 40, 50)
>>>print(20 in numbers)
True
>>>print(100 in numbers)
False
```

See another program (Figure 10.7) that checks if a string element is present in tuple or not .

A tuple can hold another tuple; such tuples are called nested tuples. The membership operator **in** Python checks for the presence of an entire

```
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> weekdays = ('Mon', 'Tue', 'Wed', 'Thu', 'Fri')
>>> day = 'Sun'
>>> if day not in weekdays:
...     print("It is a weekend")
...
It is a weekend
Ln: 14 Col: 0
```

Figure 10.7: Membership checking in tuple

element within a tuple, not parts of it. When tuples are nested, membership is tested against each whole sub-tuple rather than the individual items inside it.

Let us consider an example. Suppose we create a tuple named `total_score`, where each element represents the final score of a student in three components: CE (Continuous Evaluation), PE (Practical Evaluation), and TE (Terminal Evaluation). Here we have created tuple with three members.

```
>>>total_score = ((20, 40, 60),(18, 39, 45),(17, 34, 40))
>>>print((20, 40, 60) in total_score)
True
>>>print(40 in total_score)
False
```

The expression `(20, 40, 60) in total_score` returns `True` because this exact tuple is present as one of the elements. While the expression `40 in total_score` returns `False` because Python looks for 40 as a complete element, not as a value inside one of the sub-tuples.



Let us do

1. Write a Python program to check whether the number 25 is present in the tuple: `nums = (10, 15, 20, 25, 30, 35)`. If it exists, print "Found!", otherwise print "Not Found!"
2. What will be the output of the following code? Why?

```
animals = ('Cat', 'Dog', 'Rabbit')
print('dog' in animals)
print('Dog' in animals)
```

10.6.2 Conversions and Assignment

A key feature of Python's data structures is the ability to convert between them. This is often necessary when we need to switch between the mutability (changeability) of a list and the immutability (security/stability) of a tuple. We use the built-in functions `tuple()` and `list()` to perform these conversions.

List to Tuple

The **`tuple()`** function takes the list elements and creates a new immutable tuple. Following code demonstrates this.

```
>>>my_list = [1,2, 3]
>>>my_tuple = tuple(my_list)
>>>print(my_tuple)
(1, 2, 3)
```

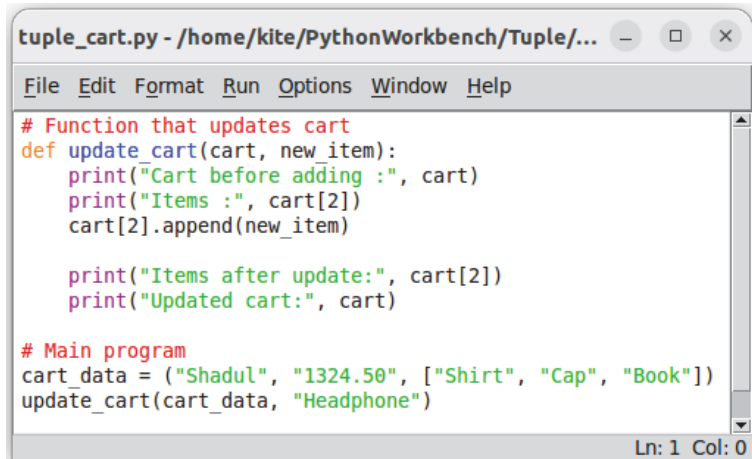

Tuple to List

The **list()** function takes the tuple elements and creates a new mutable list. In the example program given below, a tuple is first created with a few values. When we pass this tuple to **list()**, Python generates a new list containing the same elements.

```
>>>my_tuple = (4,5,6)
>>>my_list = list(my_tuple)
>>>print(my_list)
[4, 5, 6]
```

Tuples Containing Mutable Members (Lists)

A tuple can hold any type of data and this includes lists. Even though the tuple itself is immutable, the list inside it can be changed. See the program given in Figure 10.8 (a), it defines a function named **update_cart()** that updates a shopping cart stored as a tuple. The tuple contains three elements customer name, total amount and a list of items (the only mutable part).



```
tuple_cart.py - /home/kite/PythonWorkbench/Tuple/...
File Edit Format Run Options Window Help

# Function that updates cart
def update_cart(cart, new_item):
    print("Cart before adding :", cart)
    print("Items :", cart[2])
    cart[2].append(new_item)

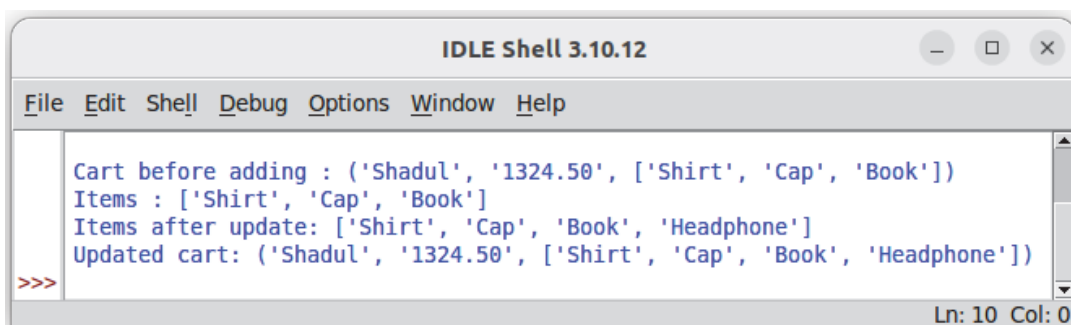
    print("Items after update:", cart[2])
    print("Updated cart:", cart)

# Main program
cart_data = ("Shadul", "1324.50", ["Shirt", "Cap", "Book"])
update_cart(cart_data, "Headphone")

Ln: 1 Col: 0
```

Figure 10.8(a): Cart updating using tuple

The function displays the cart before and after adding a new item. The statement **cart[2]** accesses the list of items and **append(new_item)** adds a new product "Headphone" to that list. The updated cart is then printed. Output of the program is shown in Figure 10.8(b).



```
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help

Cart before adding : ('Shadul', '1324.50', ['Shirt', 'Cap', 'Book'])
Items : ['Shirt', 'Cap', 'Book']
Items after update: ['Shirt', 'Cap', 'Book', 'Headphone']
Updated cart: ('Shadul', '1324.50', ['Shirt', 'Cap', 'Book', 'Headphone'])
>>>

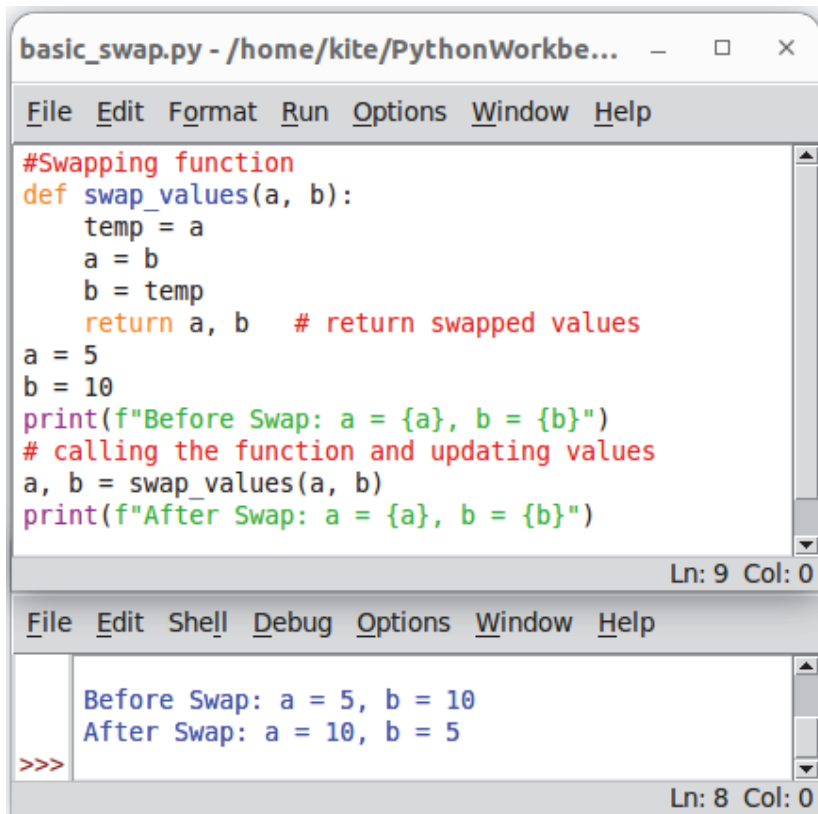
Ln: 10 Col: 0
```

Figure 10.8(b): Output of the cart updating operation

Tuple Assignment (Swapping Variables)

In Python, tuple assignment (or sequence unpacking) provides an elegant way to assign multiple values to multiple variables simultaneously. It offers a clean and concise method for swapping values or unpacking sequences, without the need for a temporary

variable. With conventional assignment, swapping the values of two variables (*a* and *b*) requires an additional temporary variable (*temp*).

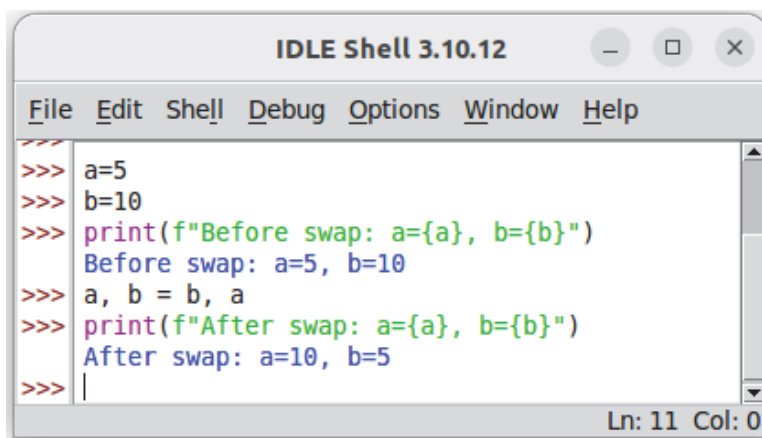


```

basic_swap.py - /home/kite/PythonWorkbe...
File Edit Format Run Options Window Help
#Swapping function
def swap_values(a, b):
    temp = a
    a = b
    b = temp
    return a, b # return swapped values
a = 5
b = 10
print(f"Before Swap: a = {a}, b = {b}")
# calling the function and updating values
a, b = swap_values(a, b)
print(f"After Swap: a = {a}, b = {b}")
Ln: 9 Col: 0
File Edit Shell Debug Options Window Help
Before Swap: a = 5, b = 10
After Swap: a = 10, b = 5
>>>
Ln: 8 Col: 0

```

Figure 10.9 (a): Swapping of values between two variables using a temporary variable



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> a=5
>>> b=10
>>> print(f"Before swap: a={a}, b={b}")
Before swap: a=5, b=10
>>> a, b = b, a
>>> print(f"After swap: a={a}, b={b}")
After swap: a=10, b=5
>>> |
Ln: 11 Col: 0

```

Figure 10.9(b): Swapping using tuple assignment

The example in Figure 10.9(a) shows how to swap the values of two variables using a temporary variable. This traditional method requires an extra step to store one value before completing the swap. In contrast, tuple assignment performs the swap in a single line of code. As shown in Figure 10.9(b), Python swaps the values of two variables using tuple packing and unpacking. Initially, *a* is 5 and *b* is 10. The statement *a, b = b, a* exchanges their values without using a temporary variable by packing the values into a tuple and then unpacking them in reverse order. After swapping, *a* becomes 10 and *b* becomes 5.

10.6.3 More Functions

Tuples have only two built-in methods for inspecting their contents: `count()` and `index()`. However, many of Python's general built-in functions can also operate on tuples. A comprehensive

list of useful functions for working with tuples is provided in Table 10.2.

Table 10.2: Common Functions on Tuples

Function	Description	Example
count()	Find out how many times a value appears in the tuple.	<pre>>>>marks = (90, 80, 90, 70, 90) >>>marks.count(90) 3</pre>
index()	Find the first position where a value appears.	<pre>>>>ai_words = ("Chatbot", "NLP", "LLM", "ANN","Chatbot") >>>ai_words.index("Chatbot") 0</pre>
len()	Returns the number of elements.	<pre>>>>items = ("pen", "book", "bag") >>>len(items) 3</pre>
max()	Returns the largest element (for numbers or alphabetically).	<pre>>>>marks = (90, 80, 90, 70, 90) >>>max(marks) 90</pre>
min()	Returns the smallest element	<pre>>>>min(marks) 70</pre>
sum()	Returns the sum of elements (if numeric).	<pre>>>>sum(marks) 420</pre>
all()	Returns True if all elements in a tuple are True (truthy). If any element is False, 0, or empty, it returns False.	<pre>>>>all(marks) True</pre>



Let us do

1. Predict the output of the following code snippet

A) `A = (10, 20, 30, 40, 50)`

`print(A[-1], A[2:4])`

B) `E = (100, 200, 300, 400, 500)`

`print(E[1], E[2:])`

2. Store the names of India's neighbouring countries in a tuple, arranged in the order of the length of the border they share with India (starting with the largest border).

Using any one of the functions listed in Table 4 (such as `min()` or `max()`), print: when arranged alphabetically (a) the first country and b) the last country from the tuple.

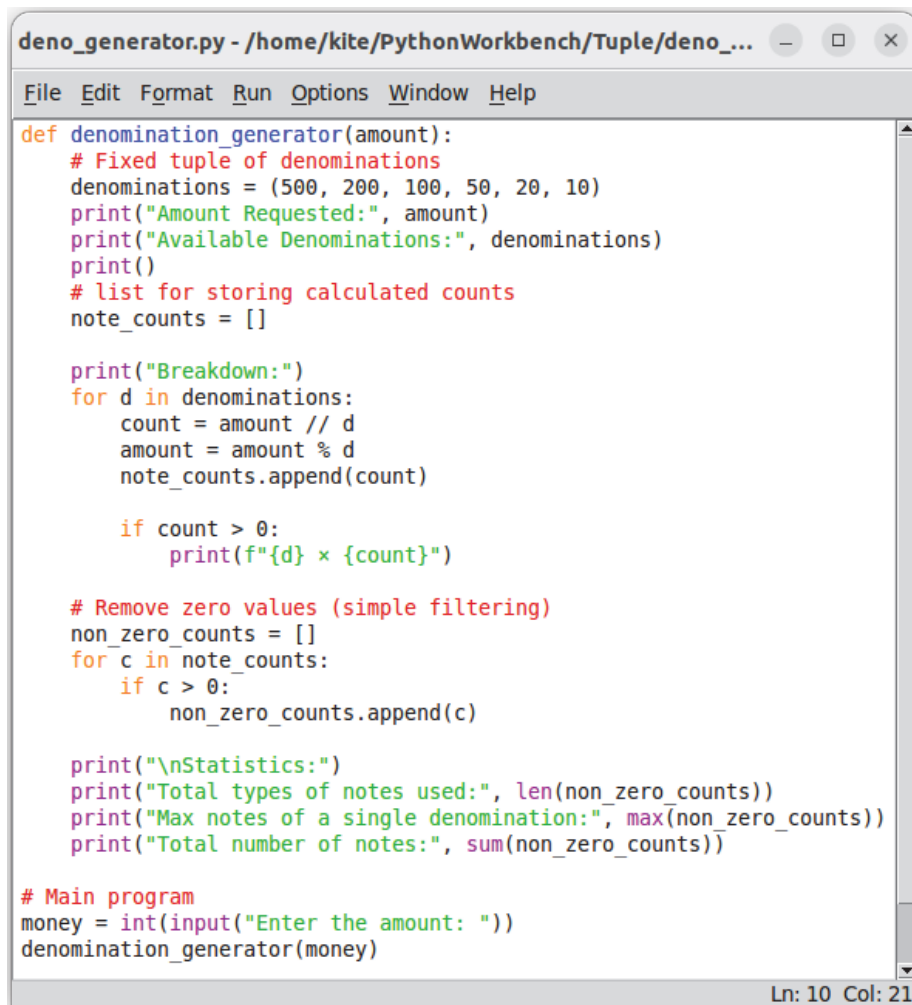
10.7 Tuple Applications

In many applications, certain pieces of data must remain unchanged once created. For example, a student's admission record—such as (ID, Name, Class), should not be modified accidentally. A tuple is ideal for representing such fixed, structured information. Here we are learning some applications using tuple.

10.7.1 Currency Denomination Analyser

Let us write a Python program that accepts an amount of money from the user and breaks it down into fixed currency denominations such as 500, 200, 100, 50, 20, and 10 rupees. The program should display how many notes of each denomination are needed to make up the given amount. After generating the breakdown, the program must also calculate and display simple statistics, including the total number of different types of notes used, the maximum number of notes of any single denomination, and the total number of notes required to represent the amount.

The following program (Figure 10.10 (a)) accepts an amount of money from the user and breaks it down into the minimum number



```
deno_generator.py - /home/kite/PythonWorkbench/Tuple/deno_...
File Edit Format Run Options Window Help

def denomination_generator(amount):
    # Fixed tuple of denominations
    denominations = (500, 200, 100, 50, 20, 10)
    print("Amount Requested:", amount)
    print("Available Denominations:", denominations)
    print()
    # list for storing calculated counts
    note_counts = []

    print("Breakdown:")
    for d in denominations:
        count = amount // d
        amount = amount % d
        note_counts.append(count)

        if count > 0:
            print(f"{d} x {count}")

    # Remove zero values (simple filtering)
    non_zero_counts = []
    for c in note_counts:
        if c > 0:
            non_zero_counts.append(c)

    print("\nStatistics:")
    print("Total types of notes used:", len(non_zero_counts))
    print("Max notes of a single denomination:", max(non_zero_counts))
    print("Total number of notes:", sum(non_zero_counts))

# Main program
money = int(input("Enter the amount: "))
denomination_generator(money)

Ln: 10 Col: 21
```

Figure 10.10(a): Program that calculates currency denomination

of currency notes. It uses **one tuple** and **one list** to complete the task. This program uses a user-defined function named `denomination_generator(amount)` to process the amount entered by the user. The fixed currency denominations are stored in a tuple because these values remain constant. As the program calculates how many notes are needed for each denomination, the results are stored in a list, which allows values to be added and changed. After generating the

breakdown, the program removes zero counts and uses the built-in functions `len()`, `max()`, and `sum()` to compute basic statistics about the notes used. The main part of the program simply accepts the amount from the user and calls the function, keeping the program structure clear and easy to follow. The output of the program is given in Figure 10.10 (b).

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Enter the amount: 24190
Amount Requested: 24190
Available Denominations: (500, 200, 100, 50, 20, 10)

Breakdown:
500 x 48
100 x 1
50 x 1
20 x 2

Statistics:
Total types of notes used: 4
Max notes of a single denomination: 48
Total number of notes: 52
>>>
Ln: 20 Col: 0

```

Figure 10.10(b): Output of the denomination generator

10.7.2 2D Point Distance Calculation

A practical use of tuples is representing fixed-size data, such as the coordinates of a point in 2D space. Since each point always has exactly two values (x and y), and these coordinates should not change accidentally, tuples are ideal. In this application, each point is stored as a tuple like (x, y)

Let us write another Python program that reads the coordinates of two points in a 2D plane from the user. The program should accept the x and y values for each point separately, store them as tuples, and then compute the Euclidean distance between the two points using the distance formula. Finally, the program should display the calculated distance along with the coordinates of both points.

The Euclidean distance between two points (x_1, y_1) and (x_2, y_2) is calculated using the formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The program is shown in Figure 10.11(a). It uses two functions to keep the code clean and easy to understand

`read_points()`: This function reads the x and y coordinates from the user and returns them as a tuple. Since a point always has two fixed values, a tuple is the best choice.

```

dist_calculator.py - /home/kite/PythonWorkbench/Tupl...
File Edit Format Run Options Window Help

import math
#Function to read points
def read_points():
    x = float(input("x: "))
    y = float(input("y: "))
    return (x, y)

#Function to calculate Euclidean distance
def distance(p1, p2):
    return math.sqrt((p2[0] - p1[0])**2 + (p2[1] - p1[1])**2)

#Main Program
print("Provide (x,y) coordinates of P1")
pt1 = read_points()
print("Provide (x,y) coordinates of P2")
pt2 = read_points()
d = distance(pt1, pt2)
print(f"Distance between {pt1} and {pt2}: ", d)

Ln: 1 Col: 0

```

Figure 10.11(a): Program to calculate distance between points

`distance(p1, p2)`: This function processes the two points and calculates the distance between them using the Euclidean distance formula. It takes two tuples as input and returns the final distance.

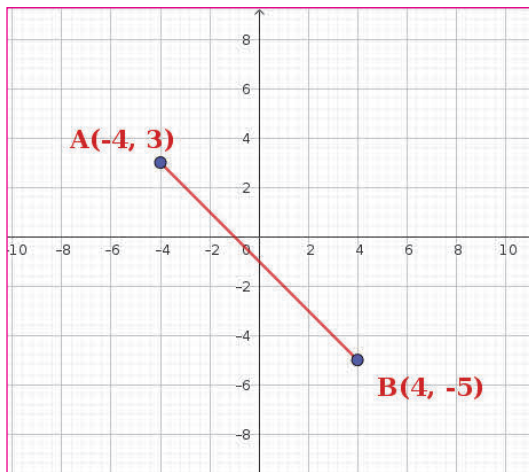


Figure 10.11(b): Graph showing the distance between two points

The main function calls the `read_points()` function twice to read the coordinates of two points. Tuple variables are used to store the coordinates (`pt1` and `pt2`). It then passes these points to the `distance(pt1,pt2)` function to compute the Euclidean distance between them. Finally, the program prints the calculated result. The graph of line connecting two points and a sample Output of the program is given in Figure 10.11(b) and 10.11(c) respectively.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Provide (x,y) coordinates of P1
x: -4
y: 3
Provide (x,y) coordinates of P2
x: 4
y: -5
Distance between (-4.0, 3.0) and (4.0, -5.0): 11.313708498984761
>>>
Ln: 13 Col: 0

```

Figure 10.11(c): Output screen of the distance calculator



Let us do

1. Write a Python program to create a list containing 14 tuples, each representing a (vehicle registration code, district name) pair of Kerala State. For example, since the registration code for Thiruvananthapuram is KL-01, the corresponding entry should be (1, "Thiruvananthapuram"). Similarly, the entry for Kozhikode should be (11, "Kozhikode").

Write the following Python functions:

`PrintDistrict()` – Accepts a district code as an input argument and prints the corresponding district name.

PrintCode() - Accepts a district name as an input argument and prints the corresponding district code.

The main program should prompt the user to choose whether to enter a district name or a district code, and then display the corresponding district code or district name by invoking the appropriate function.

2. Create a Python program to read a list of 10 tuples, each representing the coordinates of points in a two-dimensional space, from the user (recall the representation of points shown in Figure 10.11(a)). Next, read another point from the user and determine the point in the list that is closest (i.e., has the minimum distance) to the given point. You must write a function findClosest() that accepts a point (tuple) as its input argument and returns the tuple from the list that is nearest to the given point.
3. Store the details of a student (name, age, grade, state) in a tuple. Write a program in Python to unpack the tuple into separate variables and print each detail.
4. Create a tuple of bus stop names in order then: a) Print the first and last stop b) Ask the user for a stop name using input() and show its index c) Display how many stops are in the route with len()
5. Consider the following collection of books stored in a tuple:

```
books = ("When Breath Become Air", "Atomic Habits", "Snow",
         "Rubaiyat of Omar Khayyam", "The Alchemist",
         "Diary of a Wimpy Kid", "Atomic Habits")
```

Write a Python program using functions to perform the following tasks:

- a) Total Number of Books - Print the total number of books in the collection.
- b) Book Repetition - Ask the user to enter a book title and print how many times that title appears in the collection.
- c) Position on the Shelf - Ask the user for a book title and print its first position (index) on the shelf.
- d) Alphabetical Order - Find and print the titles of the first and last books when arranged alphabetically.
- e) Check Availability - Ask the user for a book title and check whether it is available in the collection.

In this chapter, we explored tuples, an important immutable sequence type in Python. We learned how tuples are created, how they differ from lists through immutability, and why this property makes them useful for storing fixed collections of values. The chapter also introduced tuple operations such as indexing, negative indexing, and slicing, which allow us to access specific elements or

portions of a tuple. We examined key tuple methods and saw how tuple packing and unpacking provide a concise way to assign and organise multiple values. Through these concepts and examples, the chapter highlighted how tuples offer a simple, reliable, and efficient way to handle structured data that should not be modified. Now let us move on to the next chapter, where we will explore two different and powerful data structures in Python - dictionaries and sets.

Dictionaries and Sets

Python provides several built-in data structures, each designed for specific types of tasks. In the previous chapters, we learned lists and tuples, which store elements in a sequential order. In this chapter, we explore two versatile data structures: dictionaries and sets. Both are widely used in real-world applications.

A **dictionary** stores data as *key-value* pairs, where each key acts like a meaningful label instead of a numeric index as in list or tuples. This makes dictionaries ideal for quickly accessing information using names, codes, or identifiers, for example, retrieving a student's grade by name or finding a product's price using its item code. A dictionary works much like a real-world dictionary, where a word (the key) helps you quickly find its meaning (the value). It can also be compared to an address book, where a person's name acts as the key and all their details—phone number, email, and address—form the value. Since dictionaries can store any type of value, including lists and even other dictionaries, they are very flexible and useful for representing structured information.

A **set**, on the other hand, is an unordered collection of unique items. Sets automatically remove duplicates and are especially useful when dealing with distinct values or when performing mathematical operations like union, intersection, and difference. They help solve problems such as finding unique words in a paragraph, identifying common students between two classes, or removing repeated entries from a list.

Key-Value Mapping in Dictionary

We can think of a dictionary as a mapping, where each key points to exactly one value. Keys are unique, so the same key cannot map to multiple values. This idea is depicted in Figure 11.1.

A Python object must be **hashable** to be used as a dictionary key. An object is considered hashable if it has a fixed internal hash value that remains the same throughout its lifetime. Hashing is the process of computing this unique integer value from the object's contents, and Python uses this hash value to quickly locate the associated dictionary entry.

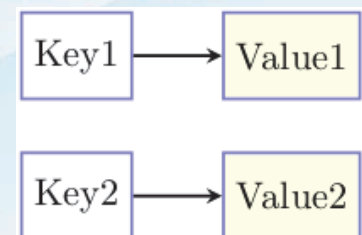


Figure 11.1: Dictionaries store items as key-value pairs.

For Python's built-in types, only immutable objects, such as integers, strings, and tuples containing only immutable elements, are hashable and can therefore be used as dictionary keys. Mutable objects like lists can change after creation, so they cannot maintain a stable hash value. For this reason, mutable types cannot be used as dictionary keys. Additionally, each key in a dictionary must be unique, ensuring that every key maps to one distinct value.

Although dictionary items are not accessed using numeric indexes, Python dictionaries preserve the order in which key-value pairs are added. They can also store any type of value, including lists and even other dictionaries. This makes dictionaries a flexible and powerful way to organize information.

For example, in a student record, the admission number can be the key and the grade its value. Since dictionaries access items using keys instead of positions, they provide fast lookup and are very useful when information needs to be retrieved quickly. A lookup refers to retrieving a value from a dictionary using its key.

Dictionaries can efficiently represent structured data such as records, symbol tables etc. Dictionaries are sometimes called associative arrays or hashes because they associate each key with a value, similar to how we use an index in a list, but here, the "index" is the key itself, which can be a string, number, or any other immutable type.



Why dictionary keys must be unique?

Behind the scenes, dictionaries use a special system called **hashing**, a technique that gives each key a unique digital signature. This digital signature helps Python quickly identify and retrieve the correct value, making dictionary lookups almost instant, much faster than searching through a list.

Hashing and digital signatures aren't just used in dictionaries; they play an important role in security and digital systems for protecting data and verifying authenticity. We can explore these concepts further as we move into advanced topics in computing, such as cryptography, database indexing, and blockchain technology.

11.1 Creating Dictionary

A dictionary can be created as a collection of key-value pairs enclosed in curly braces { }. Each key is followed by a colon : and each pair is separated by a comma.

Syntax for creating dictionary is

```
dictionary_name = {key1: value1, key2: value2, key3: value3, ...}
```

Let us begin with a simple example that stores student names with their admission numbers in a class. The following code shows how to create a dictionary for this purpose.

```
students= {
    101:"Abhilash",
    102:"Ayra",
    103:"Azim",
    104:"Brijesh",
    105:"Jaya"
}
```

Here,

- The keys are the admission numbers (101, 102, 103, 104, 105)
- The values are the names ("Abhilash", "Ayra", "Azim", "Brijesh", "Jaya").

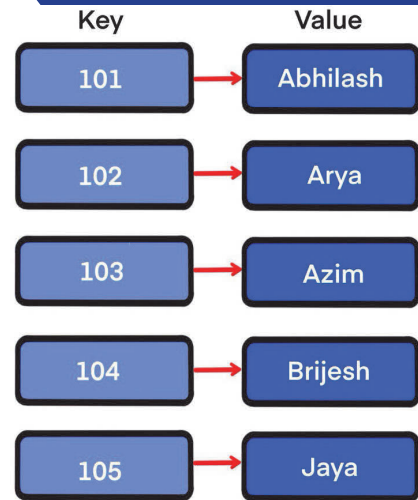


Figure 11.2: A student dictionary

Figure 11.2 illustrates how each key is mapped with their corresponding values. Another example of a dictionary is shown below, where country names act as keys and their capitals are the corresponding values. In this case, both the keys and the values are strings.

```
capitals = {
    "India":"New Delhi",
    "Mongolia":"Ulaanbaatar",
    "France":"Paris",
    "Oman":"Muscat",
    "Japan":"Tokyo",
    "Brazil":"Brasília",
    "Mali":"Bamako"
}
```

Dictionaries can also be created using the `dict()` function. Consider the example shown below:

```
>>>software_freedom = dict(
    F0="The freedom to run the program for any purpose.",
    F1="The freedom to study how the program works and
        modify it.",
    F2="The freedom to redistribute copies to help
        others.",
    F3="The freedom to improve the program and release
        your improvements."
)
```

We can create an empty dictionary using `{}`, as `empty_dict={}`. Since it has no key-value pairs inside, printing the dictionary displays `{}`. Empty dictionaries are useful when we want to start with no data and add items later in our program.

11.2 Accessing Dictionary Elements

Dictionaries allow quick access to values based on their keys. In a dictionary, each key is linked to a specific value, and we can access

that value by placing the key inside square brackets. The standard and most widely used way to access elements from a dictionary in Python is

```
value = dictionary[key]
```

Using the students dictionary we created earlier, we can retrieve a specific student's name by using the admission number as the key

```
>>>print(students[103])
Azim
```

In the same way, we can use a string key in the capitals dictionary to get the capital of a country:

```
>>>print(capitals["Oman"])
Muscat
```

If we try to access a key that does not exist in the dictionary, Python raises a `KeyError`. The example below shows this behaviour.

```
>>> print(capitals["Bhutan"])
Traceback (most recent call last):
  File "/usr/lib/python3.10/idlelib/run.py", line 578, in runcode
    exec(code, self.locals)
  File "<pyshell#4>", line 1, in <module>
KeyError: 'Bhutan'
```

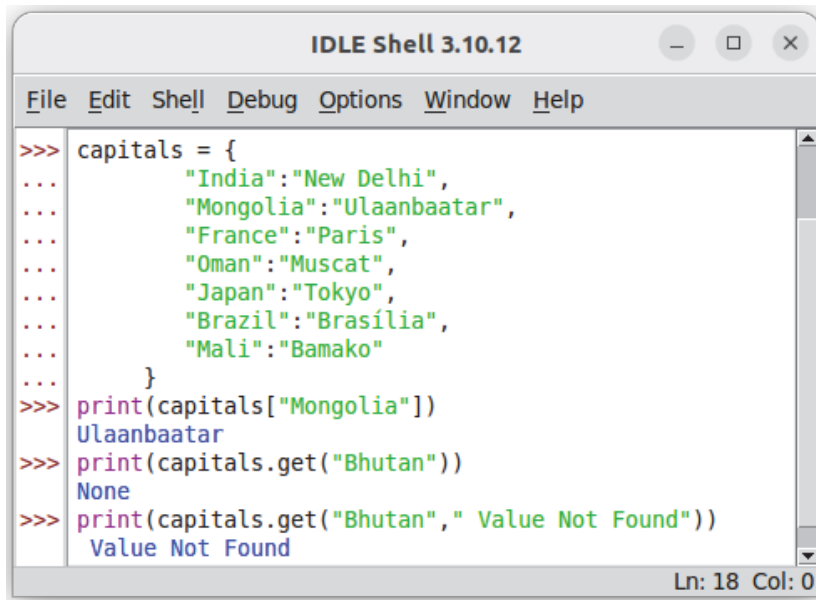


Figure 11.3: Various ways of accessing dictionary values

Another way to access dictionary values is by using the `get()` method. It works like square brackets, but with one key difference: if the key is missing, `get()` does not raise an error. Instead, it returns `None` or a default value that we specify. In the example shown in Figure 11.3, the key "Bhutan" is not present, so `get()` returns `None`, and we can also provide a custom value to print.

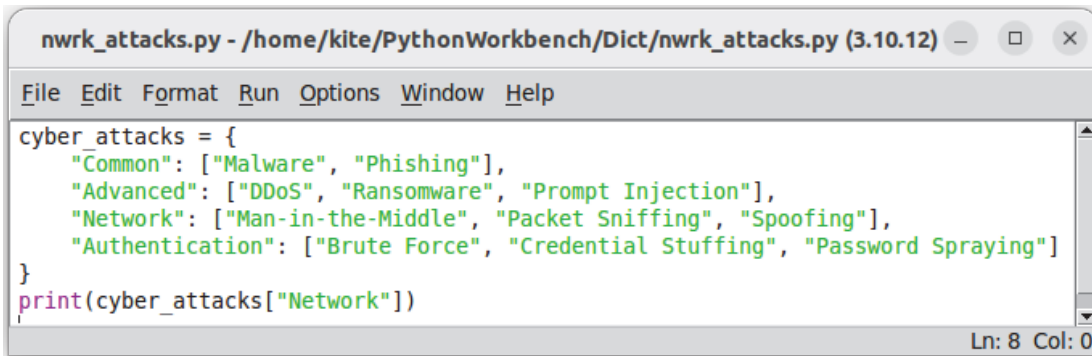
11.3 Dictionaries with Compound Values

In many real-life situations, a single key in a dictionary may need to represent more than one piece of information. For example, a subject may have several topics, a country may have multiple identifying details, or a festival and the states where it is celebrated, a movie and its cast members etc. To store such grouped information, dictionaries can use compound values, where the value associated with a key is not a single item but a collection such as a list, tuple, set, or even another dictionary. Lists are suitable when the data may change

because they are mutable, while tuples are preferred for fixed sets of items that should remain constant. Using these collections as values allows dictionaries to hold structured and meaningful data, making them a powerful tool for organising related information efficiently.

11.3.1 Dictionary with Lists as Values

A dictionary can have lists as its values. In the example shown in Figure 11.4 (a), we created a dictionary named `cyber_attacks`, where each key is a string and its value is a list of related attacks. Using a key, we can retrieve the corresponding list.

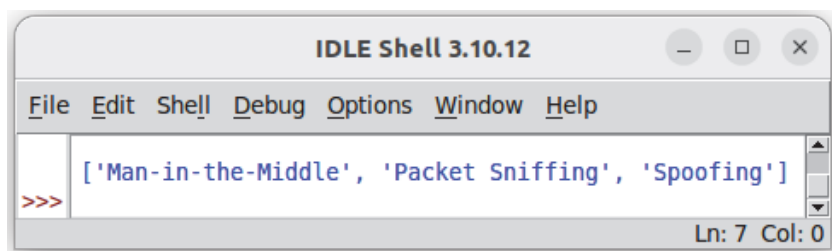


```
nwrk_attacks.py - /home/kite/PythonWorkbench/Dict/nwrk_attacks.py (3.10.12)
File Edit Format Run Options Window Help
cyber_attacks = {
    "Common": ["Malware", "Phishing"],
    "Advanced": ["DDoS", "Ransomware", "Prompt Injection"],
    "Network": ["Man-in-the-Middle", "Packet Sniffing", "Spoofing"],
    "Authentication": ["Brute Force", "Credential Stuffing", "Password Spraying"]
}
print(cyber_attacks["Network"])
Ln: 8 Col: 0
```

Figure 11.4(a): A dictionary in which each key maps to a list of values

For example, the instruction `print(cyber_attacks["Network"])` will display the list of network-based attacks as shown in Figure 11.4 (b).

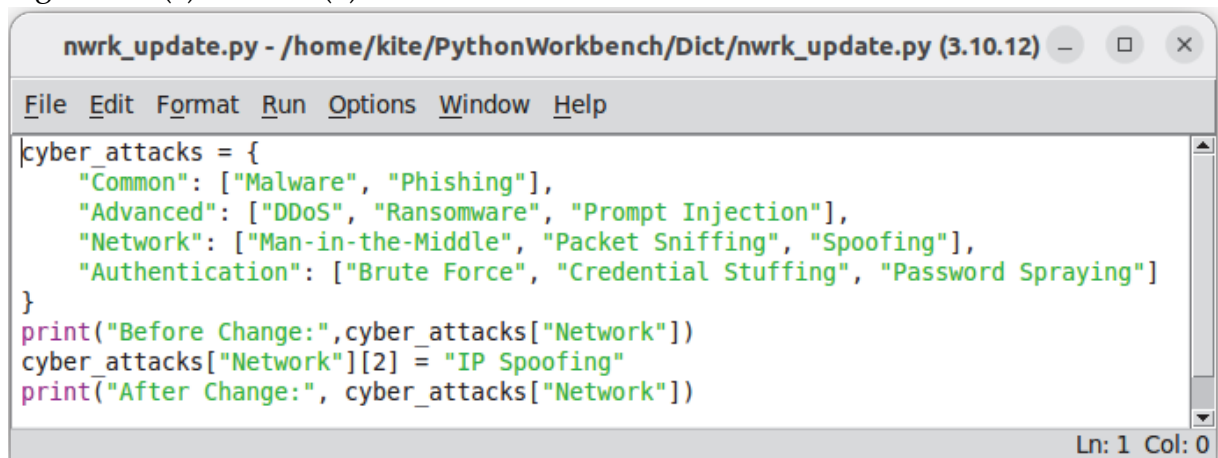
Since this dictionary stores its values as lists, we can update individual items within those lists using indexing. Since this dictionary stores its values as lists, we can update individual



```
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
['Man-in-the-Middle', 'Packet Sniffing', 'Spoofing']
>>>
Ln: 7 Col: 0
```

Figure 11.4(b) Output showing a dictionary with list values

items within those lists using indexing. For example, to correct the value "Spoofing" (stored under the key "Network") to "IP Spoofing", we modify the item at index 2 as `cyber_attacks["Network"][2] = "IP Spoofing"`. These are demonstrated in Figure 11.5(a) and 11.5(b).



```
nwrk_update.py - /home/kite/PythonWorkbench/Dict/nwrk_update.py (3.10.12)
File Edit Format Run Options Window Help
cyber_attacks = {
    "Common": ["Malware", "Phishing"],
    "Advanced": ["DDoS", "Ransomware", "Prompt Injection"],
    "Network": ["Man-in-the-Middle", "Packet Sniffing", "Spoofing"],
    "Authentication": ["Brute Force", "Credential Stuffing", "Password Spraying"]
}
print("Before Change:", cyber_attacks["Network"])
cyber_attacks["Network"][2] = "IP Spoofing"
print("After Change:", cyber_attacks["Network"])
Ln: 1 Col: 0
```

Figure 11.5(a): Updating value of a dictionary

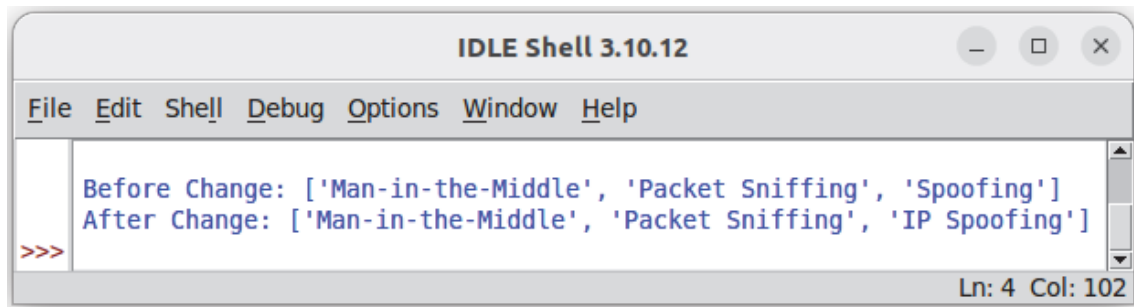


Figure 11.5(b): Output of the updated dictionary

Here, the key 'Network' refers to a list. After accessing this list, the element at index 2 is modified by assigning it a new value. We can also add new attacks to a category using `append()`.

```
cyber_attacks["Common"].append("Spam").
```

Figure 11.6(a) and 11.6(b) show the program and its output.

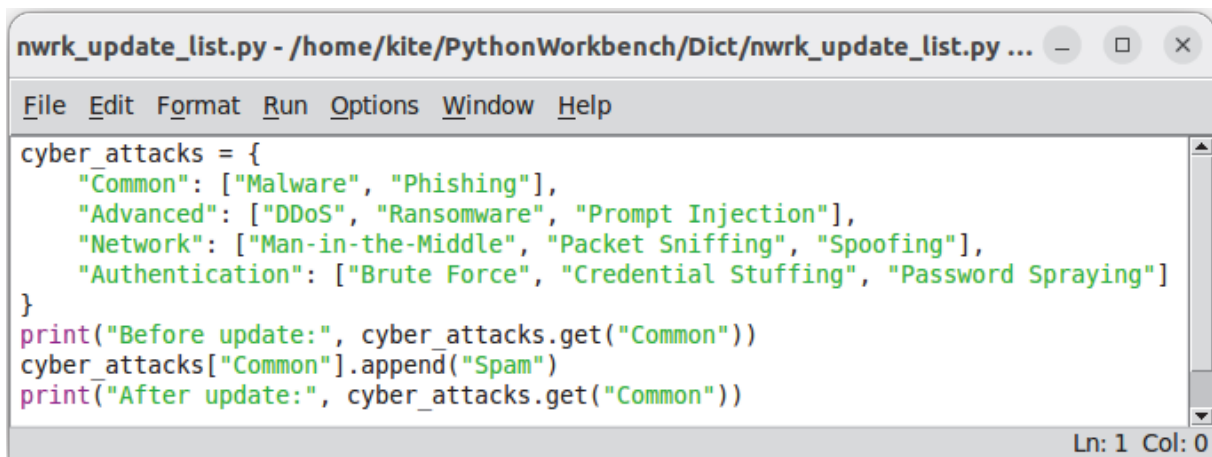


Figure 11.6(a): Adding items to a list stored inside a dictionary

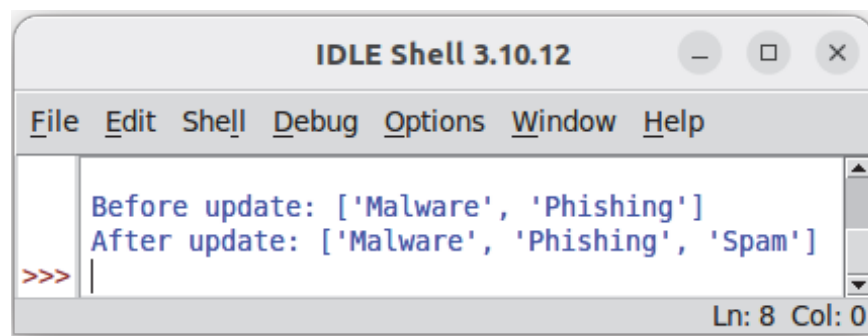


Figure 11.6(b): Output after updating the dictionary

11.3.2 Dictionary with Tuples as Values

Tuples are useful when we want to store a fixed set of items that should not be changed. Since tuples are immutable, they help protect the stored information from accidental modification. See the example given in Figure 11.7. Here, we have created a dictionary named `country_info`. Each key is a country name, and the value is a tuple containing three fixed details: the country code, the international

dialling code, and the continent. These details remain constant, so a tuple is the appropriate choice for storing them. We can retrieve specific information from a dictionary using the key and the index of the item in the tuple. Since tuples are ordered, each element can be accessed using its position (index) within the tuple. See the example below. From the `country_info` dictionary, we are accessing various elements using their respective indices:

```
country_info = {
    "India": ("IN", "+91", "Asia"),
    "United States": ("US", "+1", "North America"),
    "Japan": ("JP", "+81", "Asia"),
    "Germany": ("DE", "+49", "Europe"),
    "Nigeria": ("NG", "+234", "Africa"),
    "Brazil": ("BR", "+55", "South America"),
    "Australia": ("AU", "+61", "Australia")
}
```

Figure 11.7: creating the dictionary `country_info`

```
>>>print(country_info["India"])
('IN', '+91', 'Asia')
>>>print(country_info["India"][0])
IN
>>>print(country_info["Nigeria"][2])
Africa
```

11.3.3 Nested Dictionary

A nested dictionary is a dictionary that contains another dictionary as its value. It is useful when one piece of information has several related details. For example, The employee dictionary is an example of a nested dictionary, where each employee ID stores another dictionary containing that employee's details. In this structure, the outer dictionary uses keys such as "Emp101" and "Emp102" to identify employees. The value of each of these keys is itself a dictionary that stores related information like the employee's name, email, and phone number. See Figure 11.8 for illustration.

```
employee = {
    "Emp101": {
        "name": "Sajimon Abraham",
        "email": "sajim@example.com",
        "phone": 9876543210
    },
    "Emp102": {
        "name": "Kavitha P",
        "email": "kavitha@example.com",
        "phone": 9876501234
    }
}
```

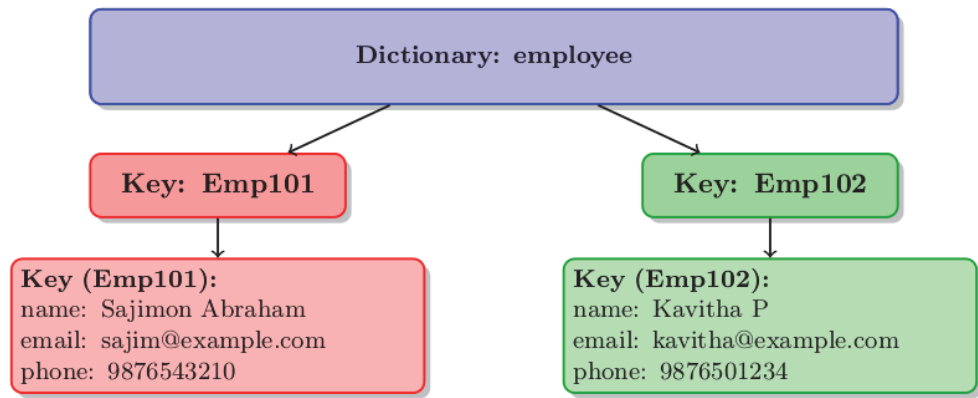


Figure 11.8: Nested Dictionary– Employee Data Example

We can retrieve various records from the dictionary as follows

```
>>>print("Name of Emp101:", employee["Emp101"]["Name"])
      Name of Emp101: Sajimon Abraham
>>>print("Phone of Emp102:", employee["Emp102"]["Phone"])
      Phone of Emp102: 9876501234
```



Let us do

Create a dictionary using the titles of your ten favourite books as keys. The value corresponding to each book should include the name of the first author, the name of the publisher, and the year of publication. Finally, print the contents of the dictionary.

11.4 Updating and Managing Dictionary Data

A dictionary is a mutable data structure, which means its contents can be changed even after it is created. Updating a dictionary refers to making any modification to its existing key-value pairs or adding new ones. There are several ways to update dictionaries in Python. Earlier, we learned how to modify lists stored as dictionary values by using operations such as appending and updating elements. Now, we will explore some additional and important methods that help us manage and update dictionaries more effectively.

11.4.1 `update()` Method

The `update()` method is useful for merging one dictionary into another or adding multiple items at once. If the key exists, its value is **updated**, and if the key does not exist, a new key-value pair is **added**. See Figure 11.9.

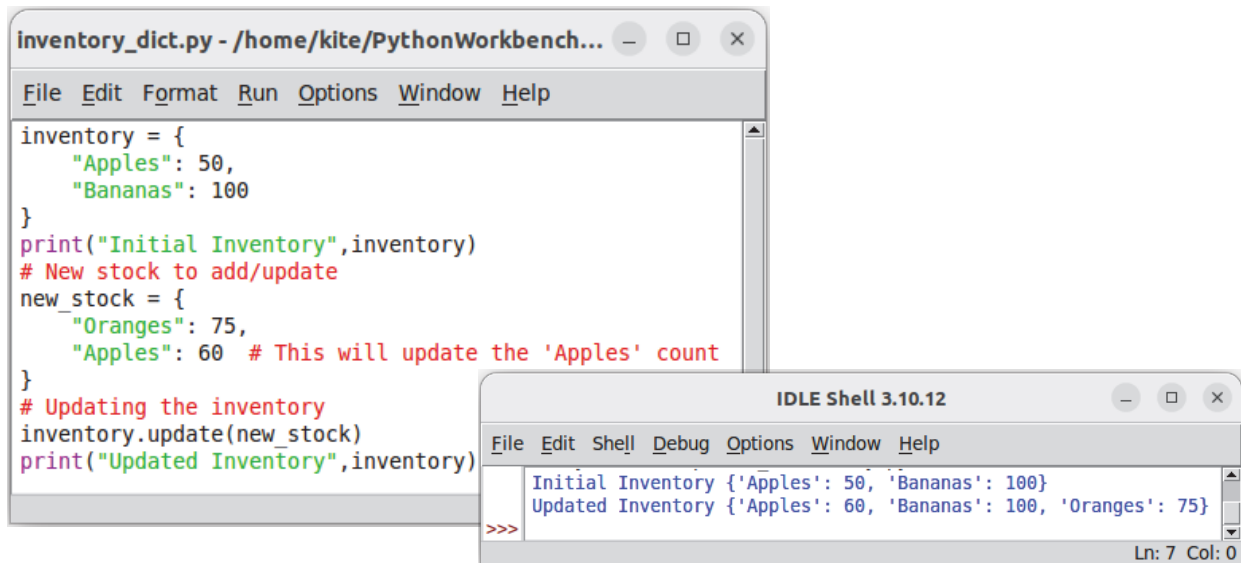


Figure 11.9: A complete dictionary is updated using the update method

11.4.2 Adding New Key-Value Pair

Assigning a value to a non-existent key automatically creates a new key-value pair in the dictionary. This is treated as an update because the dictionary's structure changes. The following statements add a new key entry (Mulberry) in to the dictionary.

```

>>>inventory = {
    "Apples": 60,
    "Bananas": 100,
    "Oranges": 75
}

>>>inventory["Mulberry"] = 180
>>>print(inventory)
{'Apples': 60, 'Bananas': 100, 'Oranges': 75,
 'Mulberry': 180}

```

11.4.3 Updating Complete Values

When a dictionary stores a list or tuple as a value, we can replace the entire list or tuple at once. See the below statements for example.

```

>>>student = {
    "name": "Aisha",
    "scores": [85, 90, 92]
}

>>>student["scores"] = [95, 98, 100]
>>>print(student)
{'name': 'Aisha', 'scores': [95, 98, 100]}

```

11.4.4 Removing a Key-Value Pair

The **del** keyword is used to remove a specific key and its associated value from a dictionary. For example,

```
>>>del inventory["Oranges"]
>>>print(inventory)
{'Apples': 60, 'Bananas': 100}
```

If the specified key does not exist in the dictionary, using the `del` keyword will raise a `KeyError`.

11.5 Dictionary Functions and Operations

Python provides several methods to remove, update, and retrieve information from dictionaries.

Some of the commonly used functions related with dictionary are given in Table 11.1.

Table 11.1: Commonly used functions in dictionary

Functions	Remarks
<code>keys()</code>	returns a list-like view of all keys in the dictionary.
<code>values()</code>	returns a list-like view of all values in the dictionary.
<code>items()</code>	returns key-value pairs from the dictionary as tuples.
<code>dictionary.update({key: value})</code>	adds a new key-value pair or updates the value of an existing key.
<code>dictionary.pop(key)</code>	removes the specified key from the dictionary and returns its value.
<code>dictionary.popitem()</code>	removes and returns the last inserted key-value pair from the dictionary.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>>
>>> gadgets_inventory = {
...     "Smartphone": 50,
...     "Smartwatch": 30,
...     "Tablet": 20,
...     "Laptop": 15,
...     "Earbuds": 25,
...     "Drone": 7
... }
>>>
>>> print("Available Products: ", gadgets_inventory.keys())
Available Products: dict_keys(['Smartphone', 'Smartwatch', 'Tablet', 'Laptop',
'Earbuds', 'Drone'])
>>>
>>> print("Available Stock: ", gadgets_inventory.values())
Available Stock: dict_values([50, 30, 20, 15, 25, 7])
>>>
>>> print("Itemwise Stock: ", gadgets_inventory.items())
Itemwise Stock: dict_items([('Smartphone', 50), ('Smartwatch', 30), ('Tablet',
20), ('Laptop', 15), ('Earbuds', 25), ('Drone', 7)])
>>>
>>> gadgets_inventory.update({"Camera": 12})
>>> gadgets_inventory["Smartphone"] = 45
>>>
>>> print("Updated Stock: ", gadgets_inventory.items())
Updated Stock: dict_items([('Smartphone', 45), ('Smartwatch', 30), ('Tablet',
20), ('Laptop', 15), ('Earbuds', 25), ('Drone', 7), ('Camera', 12)])
>>>
Ln: 27 Col: 0

```

Figure 11.10: Illustration of commonly used dictionary methods

Consider a dictionary that keeps inventory of gadgets available in a store. Based on this we are demonstrating the use of different functions (see Figure 11.10).

11.6 An Application of Dictionary

Dictionaries are widely used in real-world programming because they store data in a clear and organised key-value format. They help manage structured information efficiently, making retrieval quick and meaningful. From handling student records to storing configuration settings and organising datasets, dictionaries play a crucial role in many applications. In this section, we explore a simple and interesting application developed using a Python dictionary.

Shopping Cart Application

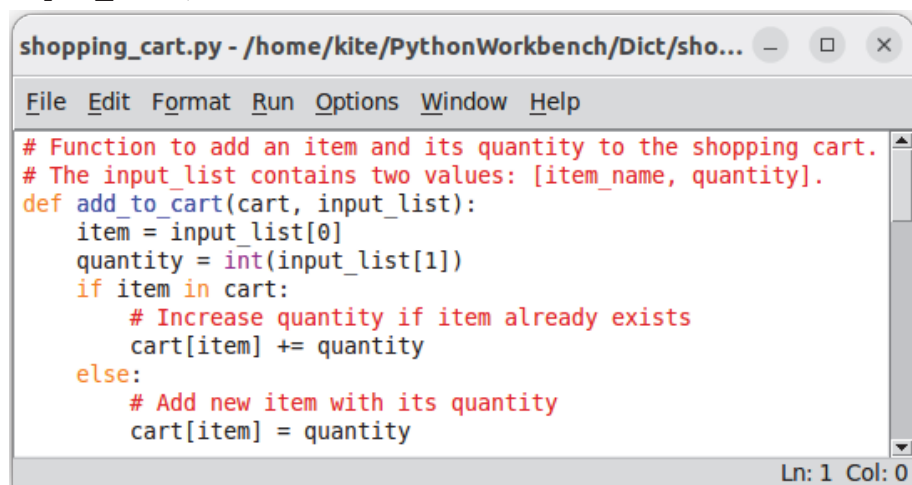
A shopping cart is a simple real-world example in which items and their quantities can be stored and processed efficiently. By using a dictionary, we can ensure that each item is listed only once, while its quantity can be updated whenever the user adds the same item again.

Let us develop a Python program that allows a user to add items and their quantities to a shopping cart. If an item is entered more than once, the program should update its quantity instead of adding a duplicate entry. After collecting all the inputs, the program should display the total number of different items in the cart, the total quantity of all products, and a formatted list of all items along with their respective quantities

The program is organised using functions, which makes the code clear, readable, and easy to maintain. After collecting all inputs, finally the program displays a detailed summary of the cart, including the total number of different items, the overall quantity of products, and a formatted list of all items in the cart. Let us see how different functions are used in the program.

`add_to_cart(cart, input_list)`

This function (shown in Figure 11.11(a)) is responsible for adding a new item to the shopping cart. The user's input, which includes the item name and quantity, is received as a two-element list.



```
shopping_cart.py - /home/kite/PythonWorkbench/Dict/sho...
File Edit Format Run Options Window Help

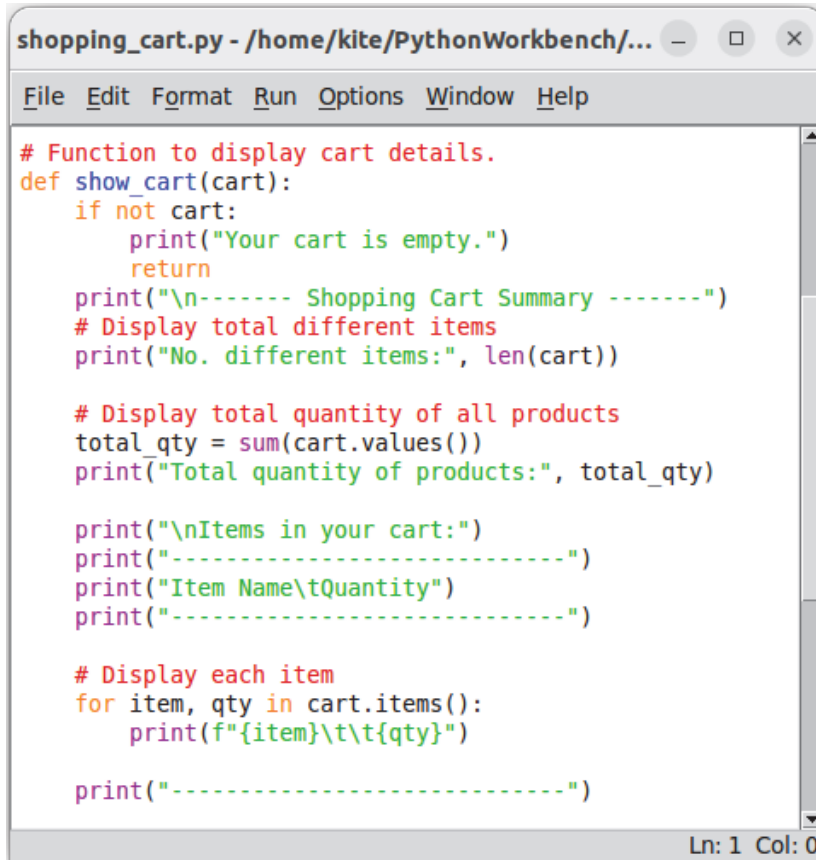
# Function to add an item and its quantity to the shopping cart.
# The input_list contains two values: [item_name, quantity].
def add_to_cart(cart, input_list):
    item = input_list[0]
    quantity = int(input_list[1])
    if item in cart:
        # Increase quantity if item already exists
        cart[item] += quantity
    else:
        # Add new item with its quantity
        cart[item] = quantity

Ln: 1 Col: 0
```

Figure 11.11(a): Function to add items to the cart

- If the item already exists in the dictionary, the function increases its quantity.
- If the item does not exist, it adds the item as a new key and stores the quantity as its value.

This ensures that the shopping cart remains updated each time the user enters an item.



```
shopping_cart.py - /home/kite/PythonWorkbench/...
File Edit Format Run Options Window Help

# Function to display cart details.
def show_cart(cart):
    if not cart:
        print("Your cart is empty.")
        return
    print("\n----- Shopping Cart Summary -----")
    # Display total different items
    print("No. different items:", len(cart))

    # Display total quantity of all products
    total_qty = sum(cart.values())
    print("Total quantity of products:", total_qty)

    print("\nItems in your cart:")
    print("-----")
    print("Item Name\tQuantity")
    print("-----")

    # Display each item
    for item, qty in cart.items():
        print(f"{item}\t{qty}")

    print("-----")
Ln: 1 Col: 0
```

11.11(b): Function to display the shopping cart

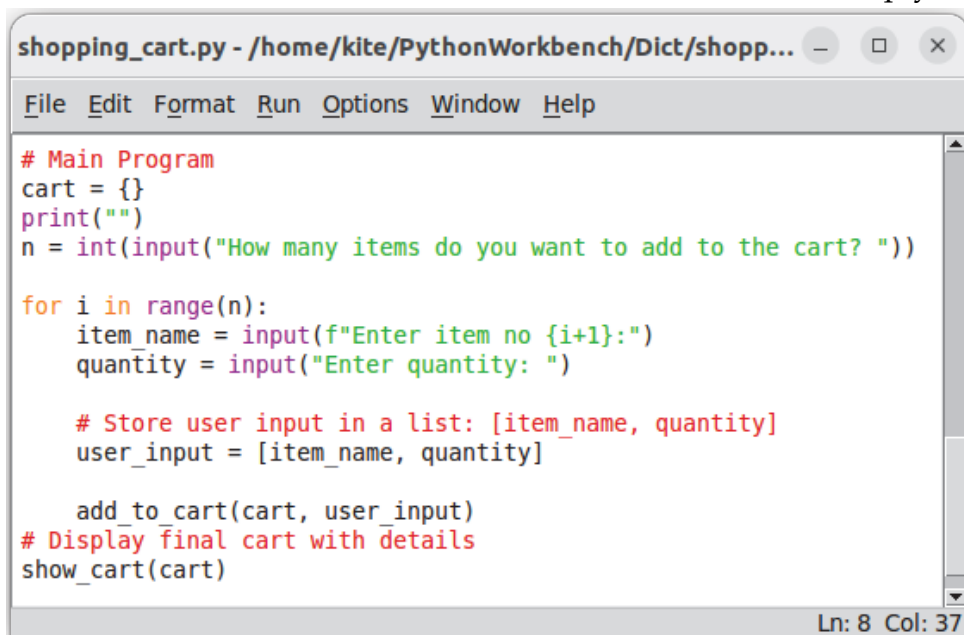
show_cart(cart)

The function shown in Figure 11.11(b) displays a complete summary of the shopping cart. It calculates and shows:

- Number of different items,
- Total quantity of all products combined,
- A tabular list of each item along with its corresponding quantity.

This provides the user with a clear overview of the contents of the cart.

The main program, shown in Figure 11.11(c), begins by creating an empty dictionary named



```
shopping_cart.py - /home/kite/PythonWorkbench/Dict/shopp...
File Edit Format Run Options Window Help

# Main Program
cart = {}
print("")
n = int(input("How many items do you want to add to the cart? "))

for i in range(n):
    item_name = input(f"Enter item no {i+1}: ")
    quantity = input("Enter quantity: ")

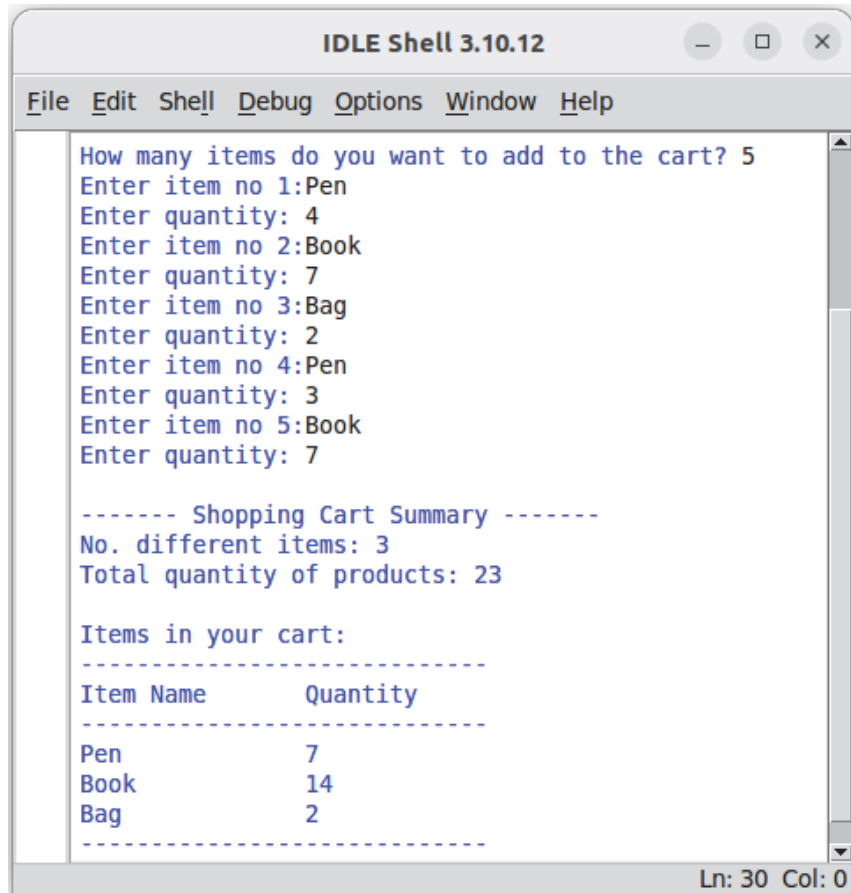
    # Store user input in a list: [item_name, quantity]
    user_input = [item_name, quantity]

    add_to_cart(cart, user_input)
# Display final cart with details
show_cart(cart)
Ln: 8 Col: 37
```

Figure 11.11(c): Main Program that invokes different functions

cart. It then asks the user how many items they want to add. Inside a loop, the program collects the item name and quantity for each entry. These details are passed to the add_to_cart() function, which updates the cart accordingly.

After all inputs have been processed, the program calls `show_cart()` to display the complete summary of the shopping cart. The output of the shopping cart application is given in Figure 11.11(d).



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
How many items do you want to add to the cart? 5
Enter item no 1:Pen
Enter quantity: 4
Enter item no 2:Book
Enter quantity: 7
Enter item no 3:Bag
Enter quantity: 2
Enter item no 4:Pen
Enter quantity: 3
Enter item no 5:Book
Enter quantity: 7

----- Shopping Cart Summary -----
No. different items: 3
Total quantity of products: 23

Items in your cart:
-----
Item Name      Quantity
-----
Pen            7
Book           14
Bag            2
-----
Ln: 30 Col: 0

```

Figure 11.11(d): Output of shopping cart application



Let us do

1. Write a Python program to work with a dictionary that stores words and their synonyms. Create a dictionary named `synonyms`, where each key is a word and each value is a list of its synonyms. Structure the solution using separate functions: one function to create the dictionary with at least ten words, another to display all the words (keys), all synonym lists (values), and the synonyms of any one word, and a third function to update the dictionary by adding a new synonym to an existing word and then adding a completely new word with its list of synonyms. The main program should call these functions in order and display the final updated dictionary.
2. Write a Python program that uses a dictionary to store information about countries and their important geographical features, such as rivers, mountains, deserts, and seas. Create a dictionary in which each key represents a country, and the corresponding value is a list containing: one major river, one major mountain, one major desert, one major sea

Organize the program using the following functions:

- a) A function to create and return the dictionary
- b) A function to print all the country names (keys)
- c) A function to print all the lists of geographical features (values)
- d) A function to print all key-value pairs in a clear and readable format.

The main program should invoke these functions in sequence and display the complete information.

3. Create a dictionary to keep track of the marks of 10 students in 3 subjects. The tuple (student name, exam name) shall form the key, and a list of 3 integers (marks) of mathematics, physics and chemistry are the values. The options for exam names are Term 1, Term 2 and Term 3. The user must have options to:
 - Give a student name, exam name, and a list consisting of the marks of three subjects as input and add them to the dictionary.
 - Give a student name and exam name as input to display the marks acquired by the student in each subject in the particular exam.

11.7 Set

A set is a collection of unique elements in Python. Like lists, tuples, and dictionaries, sets form an important part of Python's built-in data structures. A set is an **unordered** collection, meaning its elements do not have fixed positions and cannot be accessed using indexes. One of the primary uses of a set is to automatically remove duplicate values, since every element in a set must be unique. Set elements must also be immutable types such as numbers, strings, or tuples. Sets are useful when we want to store items without repetition or when we want to perform mathematical operations such as union, intersection, and difference.

11.7.1 Creation of Set

A set in Python is usually created by writing its elements inside curly braces {}, with each element separated by a comma. However, it is important to note that {} by itself does not create an empty set; it creates an empty dictionary. To create an empty set, Python uses the function `set()`. The basic syntax of a set with elements is:

```
my_set = {element1, element2, element3}
```

Consider the given examples. A set created from integers automatically removes duplicates: Sets in Python automatically remove duplicate values and do not preserve any particular order.

```
>>>numbers = {10, 20, 30, 10}
```

Although 10 appears twice, the set keeps only one copy.

```
>>>print(numbers)
{10, 20,30}
```

Now let us see another example.

```
>>>random_num = {7, 9, 11, 10, 4, 10, 20, 3, 6, 11}
>>>print(random_num)
{3, 4, 6, 7, 9, 10, 11, 20}
```

Here, the numbers 10 and 11 appear twice. The set removes the duplicate and prints all unique values in an unordered form.

Similarly, sets can be created using strings:

```
>>>browsers = {"Chrome", "Firefox", "Safari",
               "Edge", "Opera", "Brave"}
>>>print(browsers)
{'Chrome', 'Opera', 'Edge', 'Brave', 'Safari',
 'Firefox'}
```

Consider the code given below. Here, we are creating a set using a string. When the string is converted into a set, all repeating letters are automatically removed. This happens because we used the `set()` method, which converts the given value into a set of unique elements. Since sets are unordered collections, we cannot predict the exact order in which the elements will appear in the output. For example:

```
>>>letters = set("balloon")
>>>print(letters)
{'b', 'n', 'a', 'l', 'o'}
```

In this output, each letter appears only once, but the order may vary each time the program is executed.

Creating an Empty Set

We have learned `{}` creates an empty dictionary, not an empty set.

To make an empty set:

```
>>>empty_set = set()
>>>print(empty_set)
set()
```

Here, `set()` represents an empty set; an empty set is a set that contains no elements.

11.7.2 Set Functions and Operations

Python provides several useful methods to perform operations on sets. These methods help in adding elements, removing elements, and performing mathematical set operations such as union, intersection, and difference. Since sets store only unique items and do not maintain order, these methods are especially helpful in managing and combining data efficiently.

Consider the following sets, which we will use to understand different set methods listed in Table 11.2.

```
>>>a = {1,2,3}
>>>b = {3,4,5}
>>>lng1 = {"Python", "C", "Java"}
>>>lng2 = {"JavaScript", "Python", "Go"}
```

Table 11.2: Common set functions and operations

Function/Operator	Remarks	Examples
union() or	Combines all unique elements from both sets.	<pre>>>>a b {1, 2, 3, 4, 5} >>>lng1.union(lng2) {'Go', 'Java', 'JavaScript', 'C', 'Python'}</pre>
intersection() or &	Returns elements common to both sets.	<pre>>>>a&b {3} >>>lng1.intersection(lng2) {'Python'}</pre>
difference() or -	Elements in a set that are not in set b.	<pre>>>>a-b {1, 2} >>>lng1.difference(lng2) {'C', 'Java'}</pre>
add()	Adds a single new element to a set.	<pre>>>>a.add(6) >>>a {1, 2, 3, 6}</pre>
remove()	Removes an element from the set. (error if not present).	<pre>>>>a.remove(3) >>>a {1, 2, 6}</pre>
discard()	Removes an element from the set (no error if missing).	<pre>>>>a.discard(10)</pre>
clear()	Removes all elements from the set.	<pre>>>>a.clear() >>>print(a) set()</pre>

Sets are especially useful when we need to find common items or unique items—operations such as union, intersection, and difference are very efficient with sets.

11.7.3 Applications of Set

Let us create a Python program to compare the subjects chosen by two students.

```

stud_stat.py - /home/kite/PythonWorkbench/Set/stud_... - □ ×
File Edit Format Run Options Window Help
# Subjects chosen by two students
student_A = {"Maths", "Physics", "English", "Computer"}
student_B = {"Biology", "English", "Maths", "Chemistry"}

# Finding common subjects
common_subjects = student_A.intersection(student_B)

# Finding subjects unique to each student
unique_to_A = student_A.difference(student_B)
unique_to_B = student_B.difference(student_A)

print("Common Subjects:", common_subjects)
print("Subjects only Student A selected:", unique_to_A)
print("Subjects only Student B selected:", unique_to_B)
Ln: 11 Col: 0

```

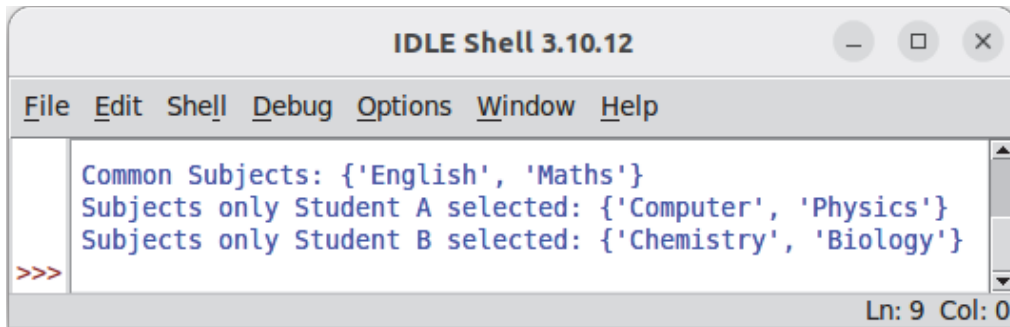
The program (see Figure 11.12) should store the list of subjects selected by Student A and Student B using sets. It should then find and display the subjects common to both students. After that, the program should identify and display the subjects that are unique to

Figure 11.12(a): Various set operations

Student A, followed by the subjects that are unique to Student B.

Let us develop another Python program that accepts a sentence from the user and displays all the unique words present in the sentence. The program should perform the following tasks:

- Prompt the user to enter a sentence, split the sentence into individual words, identify the unique words by removing any duplicates and display the set of unique words to the user.



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Common Subjects: {'English', 'Maths'}
Subjects only Student A selected: {'Computer', 'Physics'}
Subjects only Student B selected: {'Chemistry', 'Biology'}
>>>
Ln: 9 Col: 0

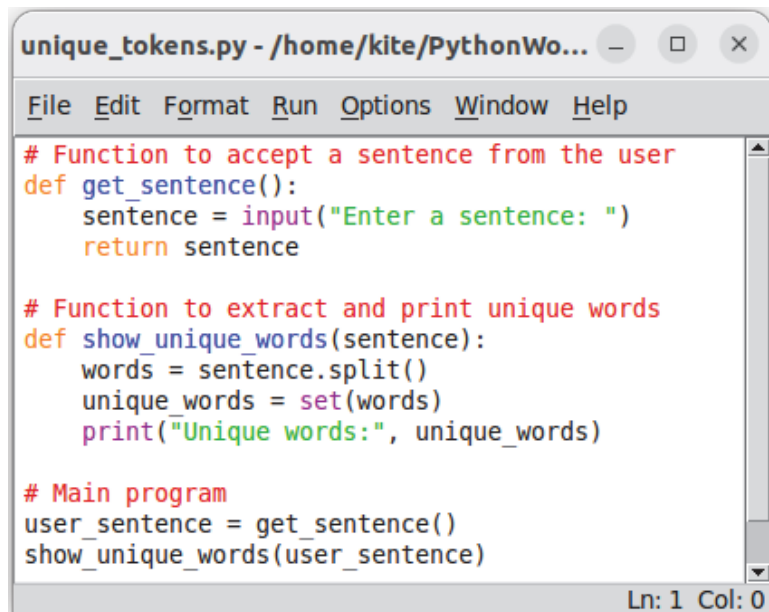
```

Figure 11.12(b): Output of program shown in Figure 11.12(a)

The program shown in Figure 11.13(a) uses two user-defined functions: `get_sentence()` and `show_unique_words()`. The main program coordinates these functions to generate the desired output. In the `show_unique_words()` function, the built-in `split()` method of Python strings is used, which separates the sentence into individual words based on spaces by default.

This list of words is then processed to identify and display only the unique ones.

The output produced by the program is shown in Figure 11.13(b).



```

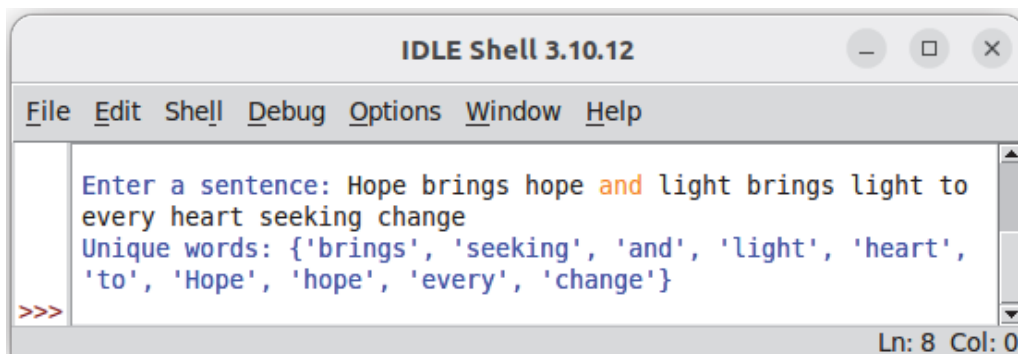
unique_tokens.py - /home/kite/PythonWo...
File Edit Format Run Options Window Help
# Function to accept a sentence from the user
def get_sentence():
    sentence = input("Enter a sentence: ")
    return sentence

# Function to extract and print unique words
def show_unique_words(sentence):
    words = sentence.split()
    unique_words = set(words)
    print("Unique words:", unique_words)

# Main program
user_sentence = get_sentence()
show_unique_words(user_sentence)
Ln: 1 Col: 0

```

Figure 11.13(a): Function that collects unique words from a sentence



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Enter a sentence: Hope brings hope and light brings light to
every heart seeking change
Unique words: {'brings', 'seeking', 'and', 'light', 'heart',
'to', 'Hope', 'hope', 'every', 'change'}
>>>
Ln: 8 Col: 0

```

Figure 11.13(b): Output of program shown in Figure 11.13(a)



Let us do

1. Write a Python program to check whether a user-entered password is valid. The program should be structured using functional decomposition by defining separate functions to perform the following tasks:
 - a) Accept the password from the user
 - b) Check whether the password contains at least one special character from the set {"@", "#", "\$", "&"}
 - c) Display whether the password is valid or invalid.

A password is considered valid if it contains: at least one uppercase letter, at least one digit and at least one special character from the given set
2. Write a Python program to identify the unique words in a sentence. Organize the solution into separate functions for:
 - (a) accepting one or more sentences from the user,
 - (b) converting the words into a set to remove duplicates, and
 - (c) displaying the resulting set of unique words.
3. A school is organizing four competitions for students: Quiz, Web Designing, Coding, and Animation. There are 7 students, and each student can participate in a maximum of three competitions of their choice. Write a Python program to analyse student participation using appropriate data structures and set operations. The program should be structured as follows: Allow the user to enter the names of the competitions in which each student is participating. Store this information in a list of tuples, where each tuple contains:
 - the name of a student, and
 - a set of competitions that the student has registered for.

Write a function that accepts the name of a competition as input and displays the names of all students who are participating in that competition.

The program should make effective use of sets to manage and analyse participation data.

The chapter explained dictionaries as collections of key-value pairs, showed how to create them, and demonstrated common operations such as accessing values, adding or updating entries, and using various standard methods. Simple applications, such as inventory information or shopping cart - highlighted how dictionaries are useful for organising and retrieving data quickly.

The chapter concluded with a brief introduction to sets, which store unique, unordered elements. It explained how to create sets, remove duplicates, and perform basic operations like union, intersection, and difference. Overall, the chapter built a basic understanding of how dictionaries and sets help manage and process data efficiently in Python.

Sorting and Searching

Sorting and searching are essential techniques for efficiently organising and retrieving data from data structures. Sorting involves arranging data in a specific order, ascending or descending, in order to make it easier to access, interpret and process it. In this chapter, we explore two widely used sorting algorithms - bubble sort and selection sort. Once a list is sorted, a more efficient searching technique called binary search can be applied, and this chapter introduces this technique. Sorting and searching are fundamental concepts in programming, as they significantly improve data processing efficiency and form the foundation for numerous advanced algorithms and applications.

12.1 Sorting

Sorting is the method of arranging elements in a specific, logical order. This logical order may be ascending or descending in the case of integer or float values, or dictionary order in the case of strings. There are many algorithms available to perform this task efficiently. In this chapter, we will discuss two of them - bubble sort and selection sort - and apply them for sorting a list.

12.1.1 Bubble Sort

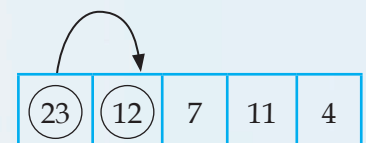
Bubble sort arranges a list of elements in ascending or descending order. It is a sorting method in which the algorithm moves through the list and compares each pair of adjacent elements. If the elements are in the wrong order, they are swapped. Now, move to the next pair and repeat until reaching the end of the list. After this pass, the largest element is placed at the end. Repeat the whole process for the remaining unsorted part of the list. Bubble sort gets its name because the larger element bubbles towards the end of the list. To understand this better, consider the following example of a number list.

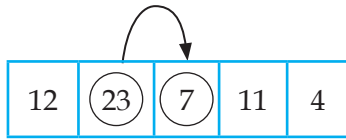
Initial list

23	12	7	11	4
----	----	---	----	---

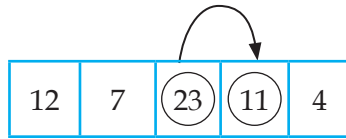
Pass 1

The first comparison results in exchanging the first two elements, 23 and 12.

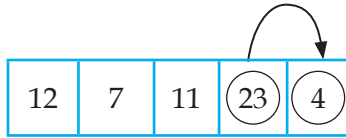




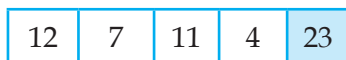
The second comparison results in exchanging the second and third elements 23 and 7 in the revised list.



The third comparison results in exchanging the third and fourth elements 23 and 11 in the revised list.

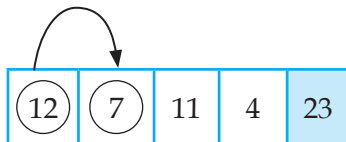


The fourth comparison results in exchanging the fourth and fifth elements 23 and 4 in the revised list.

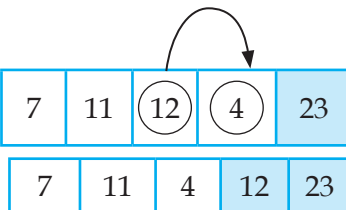
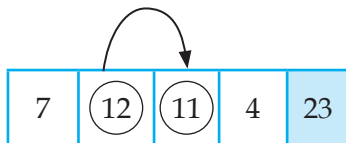


After the end of the first pass, the largest element 23 is bubbled to the last position of the list.

Pass 2

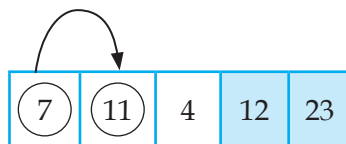


In the second pass, we consider only the four elements of the list, excluding 23.

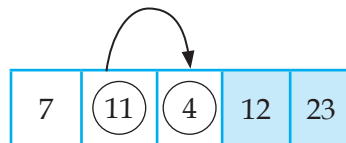


The same process is continued as in Pass 1 and as a result of it, 12 is bubbled to fourth position, which is the second largest element in the list.

Pass 3



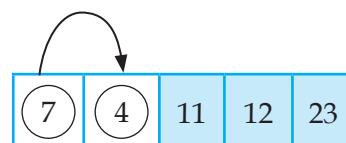
In the third pass, we consider only the three elements of the list, excluding 23 and 12.



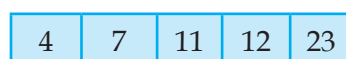
The same process is continued as in the above pass and as a result of it, 11 is bubbled to the third position.



Pass 4



In the last pass, we consider only the two elements of the list, excluding 23, 12 and 11. The same process is continued as in the above pass and as a result of it, 7 is bubbled to second position and eventually 4 is placed in the first position.

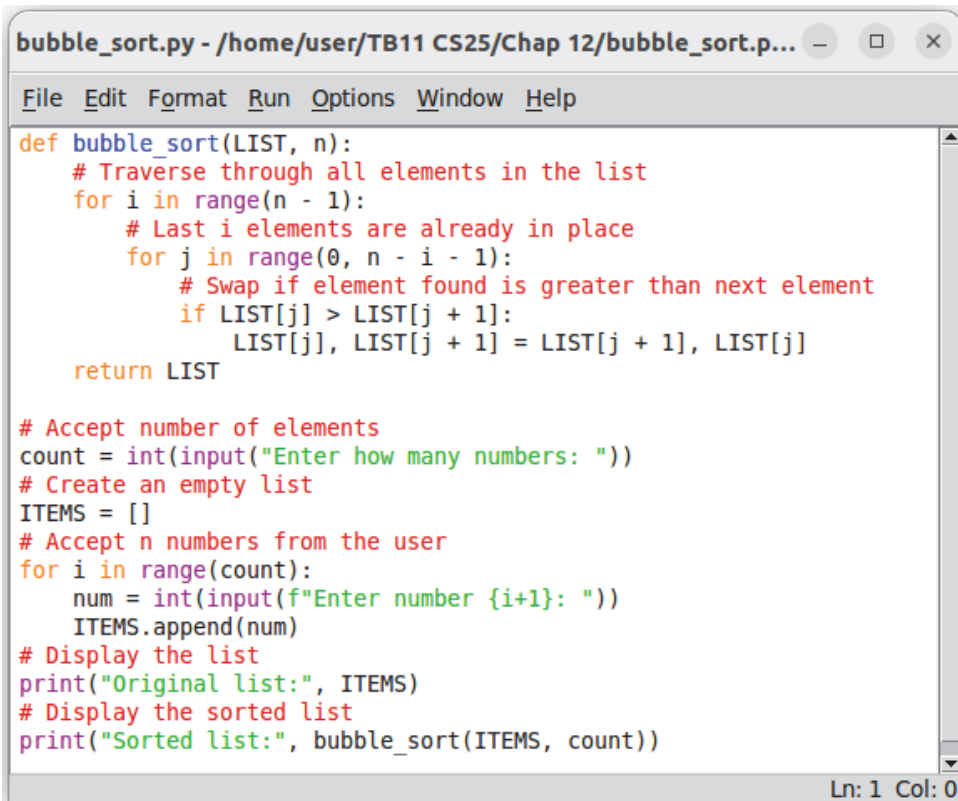


After applying the bubble sort algorithm, we obtain a sorted list of elements. To sort a list containing N elements, bubble sort requires $(N - 1)$ passes. During each pass, the number of elements to be compared reduces by one, as the largest element moves to its correct position.

Algorithm for Bubble Sort (ascending order)

- Step 1: Start.
- Step 2: Accept a value in N as the number of elements of the list.
- Step 3: Accept N elements into the list named LIST.
- Step 4: Repeat Steps 5 to 7, $(N - 1)$ times.
- Step 5: Repeat Step 6 until the second last element of the current list.
- Step 6: Starting from the first position, compare two adjacent elements in the list. If they are not in the proper order, swap the elements.
- Step 7: Revise the list by excluding the last element in the current list.
- Step 8: Print the sorted list, LIST.
- Step 9: Stop.

To improve clarity and maintainability, Python encourages dividing large code blocks into smaller, well-defined functions. This practice helps in organising the program better and allows each part to be tested and reused independently. Following this approach, the bubble sort steps (steps 5 to 7) are separated into a dedicated function, as shown in Figure 12.1.



```

bubble_sort.py - /home/user/TB11 CS25/Chap 12/bubble_sort.p...
File Edit Format Run Options Window Help
def bubble_sort(LIST, n):
    # Traverse through all elements in the list
    for i in range(n - 1):
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            # Swap if element found is greater than next element
            if LIST[j] > LIST[j + 1]:
                LIST[j], LIST[j + 1] = LIST[j + 1], LIST[j]
    return LIST

# Accept number of elements
count = int(input("Enter how many numbers: "))
# Create an empty list
ITEMS = []
# Accept n numbers from the user
for i in range(count):
    num = int(input(f"Enter number {i+1}: "))
    ITEMS.append(num)
# Display the list
print("Original list:", ITEMS)
# Display the sorted list
print("Sorted list:", bubble_sort(ITEMS, count))
Ln: 1 Col: 0

```

Figure 12.1: Program using bubble_sort()

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Enter how many numbers: 5
Enter number 1: 56
Enter number 2: 34
Enter number 3: 98
Enter number 4: 12
Enter number 5: 39
Original list: [56, 34, 98, 12, 39]
Sorted list: [12, 34, 39, 56, 98]
>>>
Ln: 1 Col: 22

```

Figure 12.2: Output of bubble_sort() function

The main function to accept numbers in a list and call bubble_sort() is also given in Figure 12.1. The output will be as in Figure 12.2.

12.1.2 Selection Sort

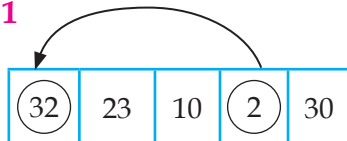
To sort a list in the ascending order, the selection sort algorithm starts by finding the smallest value in the list and placing it to the first position. At the same time, the element at the first position is shifted to the position of the smallest element. This process is then repeated to find the second smallest number and place it in the second position, and so on, until the entire list is sorted.

A pass is completed once the smallest element in the unsorted section is located and exchanged with the element at the current position. For N elements there will be $N - 1$ passes. For example, consider the following list of numbers.

Initial list

32	23	10	2	30
----	----	----	---	----

Pass 1

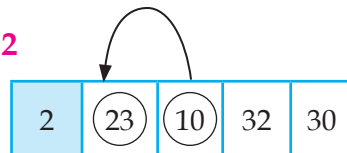


In Pass 1, the smallest element 2 is selected from the list. It is then exchanged with the first element.

After Pass 1

2	23	10	32	30
---	----	----	----	----

Pass 2

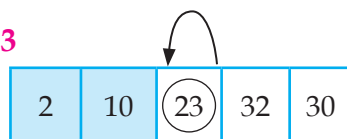


In Pass 2, excluding the first element, the smallest element 10 is selected and exchanged with the second element.

After Pass 2

2	10	23	32	30
---	----	----	----	----

Pass 3

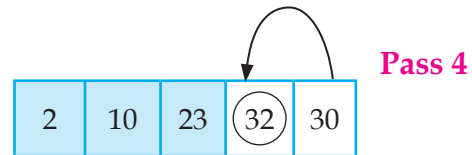


In Pass 3, ignoring the first and second elements, the smallest element 23 is selected and exchanged with the third element.

After Pass 3

2	10	23	32	30
---	----	----	----	----

In Pass 4, ignoring the 1st, 2nd and 3rd elements, the smallest element 30 is selected and exchanged with the fourth element.



Though each pass is intended for an exchange, no exchange is made in a pass if the smallest value is already in the correct location. This situation can be seen in Pass 3.



Algorithm for Selection sort

- Step 1: Start.
- Step 2: Accept a value in N as the number of elements of the list.
- Step 3: Accept N elements into the list named LIST.
- Step 4: Repeat Steps 5 to 9, (N - 1) times.
- Step 5: Assume the first element in the list as the smallest and store it in MIN and its position in POS.
- Step 6: Repeat Step 7 until the last element of the list.
- Step 7: Compare the next element in the list with the value of MIN. If it is found smaller, store it in MIN and its position in POS.
- Step 8: If the first element in the unsorted list and the value in MIN are not the same, then swap the first element with the element at position POS.
- Step 9: Revise the list by excluding the first element in the current list.
- Step 10: Print the sorted LIST.
- Step 11: Stop.

Similar to the approach used in bubble sort, the larger code sections here are divided into smaller, well-structured functions. The repeated operations of identifying the smallest element and swapping it (steps 5 to 9) are therefore grouped into a separate function, as shown in Figure 12.3.

The main function to accept numbers in a list and call `selection_sort()` is also given in Figure 12.3.

```

def selection_sort(LIST, n):
    # Traverse through all elements in the list
    for i in range(n - 1):
        # Assume first element is the minimum
        min_index = i
        # Find index of smallest element in the remaining list
        for j in range(i + 1, n):
            if LIST[j] < LIST[min_index]:
                min_index = j
        # Swap min element with the first element of unsorted part
        LIST[i], LIST[min_index] = LIST[min_index], LIST[i]
    return LIST

# Accept number of elements
count = int(input("Enter how many numbers: "))
# Create an empty list
ITEMS = []
# Accept n numbers from the user
for i in range(count):
    num = int(input(f"Enter number {i+1}: "))
    ITEMS.append(num)
# Display the list
print("Original list:", ITEMS)
# Display the sorted list
print("Sorted list:", selection_sort(ITEMS, count))
  
```

Figure 12.3: Program using `selection_sort()`

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Enter how many numbers: 5
Enter number 1: 78
Enter number 2: 45
Enter number 3: 97
Enter number 4: 34
Enter number 5: 67
Original list: [78, 45, 97, 34, 67]
Sorted list: [34, 45, 67, 78, 97]
>>>
Ln: 1 Col: 13

```

Figure 12.4: Output of `selection_sort()`

The output is shown in Figure 12.4.

12.1.3 Sorting Using Built-in Functions

Sorting in a list can be performed in Python using the built-in functions **sort()** and **sorted()**.

(a) **sort()** Function

The built-in function **sort()** in Python is used to arrange the elements of a list in a specific order either ascending or descending. It changes the original list instead of creating a new one. Note that

sort() function works only with lists.

By default, the **sort()** function arranges the elements in the ascending order. If you want to sort the list in the descending order, you can use the argument **reverse=True**.

The code in Figure 12.5 sorts the elements in the ascending order. The code in Figure 12.6 sorts the elements in the descending order.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> numbers = [5, 2, 8, 1, 9]
>>> numbers.sort()
>>> print(numbers)
[1, 2, 5, 8, 9]
>>>
Ln: 7 Col: 0

```

Figure 12.5: **sort()** function for ascending order

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> numbers = [5, 2, 8, 1, 9]
>>> numbers.sort(reverse=True)
>>> print(numbers)
[9, 8, 5, 2, 1]
>>>
Ln: 1 Col: 49

```

Figure 12.6: **sort()** function used in descending order

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> words = ['apple', 'kiwi', 'banana', 'mango']
>>> words.sort()
>>> print(words)
['apple', 'banana', 'kiwi', 'mango']
>>> words.sort(key=len)
>>> print(words)
['kiwi', 'apple', 'mango', 'banana']
>>>
Ln: 19 Col: 28

```

Figure 12.7: **sort()** function for sorting strings

Apart from numbers, **sort()** can be used to sort lists of strings, or any data type that supports comparison. The key parameter in the **sort()** function is used to specify a function that determines the sorting criteria. By default, strings are sorted alphabetically with the **sort()** function. Sorting of strings

can also be done based on word length using **key** parameter as in Figure 12.7.

(b) **sorted()** Function

This built-in function returns a new sorted list from the elements of literal collections such as list, tuple or string. It does not modify the original literal collection. Sorting can be done using **sorted()** as in Figure 12.8. Note that the original list will remain unchanged.

Similar to **sort()** function, **sorted()** function also has arguments **reverse=True** for sorting in descending order and **key=len** for sorting based on length of strings.

Even though Python provides the `sort()` and `sorted()` functions for quick and efficient sorting, it is important to study the algorithms behind these operations. Learning algorithms like bubble sort and selection sort helps you to understand how data is processed, how efficiency is measured, and why certain methods perform better in specific situations. This knowledge strengthens your programming skills and prepares you for more advanced concepts.

```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> numbers = [5, 2, 8, 1, 9]
>>> sorted_numbers = sorted(numbers)
>>> print(sorted_numbers)
[1, 2, 5, 8, 9]
>>> print(numbers)
[5, 2, 8, 1, 9]
>>>
>>> sorted_numbers = sorted(numbers, reverse=True)
>>> print(sorted_numbers)
[9, 8, 5, 2, 1]
>>>
>>> words = ['apple', 'kiwi', 'banana', 'mango']
>>> sorted_words = sorted(words, key=len)
>>> print(sorted_words)
['kiwi', 'apple', 'mango', 'banana']
>>>
Ln: 25 Col: 19

```

Figure 12.8: `sorted()` function

12.2 Searching

Searching is the process of finding the location of a given element within a collection of data, such as a list, array, or database. It helps us to determine whether a particular item is present and if present, its location. The most frequently used search algorithms are linear search and binary search. In linear search, each element is checked one by one, whereas in binary search, searching is done efficiently on sorted elements.

Efficient searching methods help to improve the performance of programs while dealing with large amounts of data.

12.2.1 Binary Search

We have discussed linear search in detail in Chapter 9. The linear search algorithm is simple and convenient to use for small lists. However, when the list is large and requires repeated searches, it is better to use a more efficient searching method. If the list elements are arranged in sorted order, a faster algorithm called binary search can be used. This method greatly reduces the search time compared to linear search.

Binary search is an efficient searching technique that uses the minimum number of comparisons to locate an element in a sorted list. The algorithm begins by checking the middle element of the list. If this element matches the value being searched for, the search is complete. If the search value is smaller than the middle element, the algorithm continues the search in the first half of the list. If the search value is larger, the search proceeds in the second half. By repeatedly dividing the list into halves, binary search quickly narrows down the position of the desired element.

Let us consider the following sorted list with seven elements to illustrate the working of the binary search technique. Assume that the element to be searched is 45.

0	1	2	3	4	5	6	
21	28	33	35	45	58	61	FIRST = 0 LAST = 6

As $FIRST \leq LAST$, let us start iteration.

$$MIDDLE = (FIRST + LAST) / 2$$

$= (0 + 6) / 2 = 3$. Here $LIST[3]$ is not equal to 45 and $LIST[3]$ is less than the search element. Therefore, we take $FIRST = MIDDLE + 1 = 3 + 1 = 4$, $LAST = 6$

0	1	2	3	4	5	6	
21	28	33	35	45	58	61	

As $FIRST \leq LAST$, we start next iteration.

$MIDDLE = (FIRST + LAST) / 2 = (4 + 6) / 2 = 5$. Here $LIST[5]$ is not equal to 45 and $LIST[5]$ is greater than the search element. Therefore, we take $FIRST = 4$, $LAST = MIDDLE - 1 = 5 - 1 = 4$,

0	1	2	3	4	5	6	
21	28	33	35	45	58	61	

As $FIRST \leq LAST$, we start next iteration

$MIDDLE = (FIRST + LAST) / 2 = (4 + 4) / 2 = 4$. Here $LIST[4]$ is equal to 45 and the search terminates successfully.

0	1	2	3	4	5	6	
21	28	33	35	45	58	61	

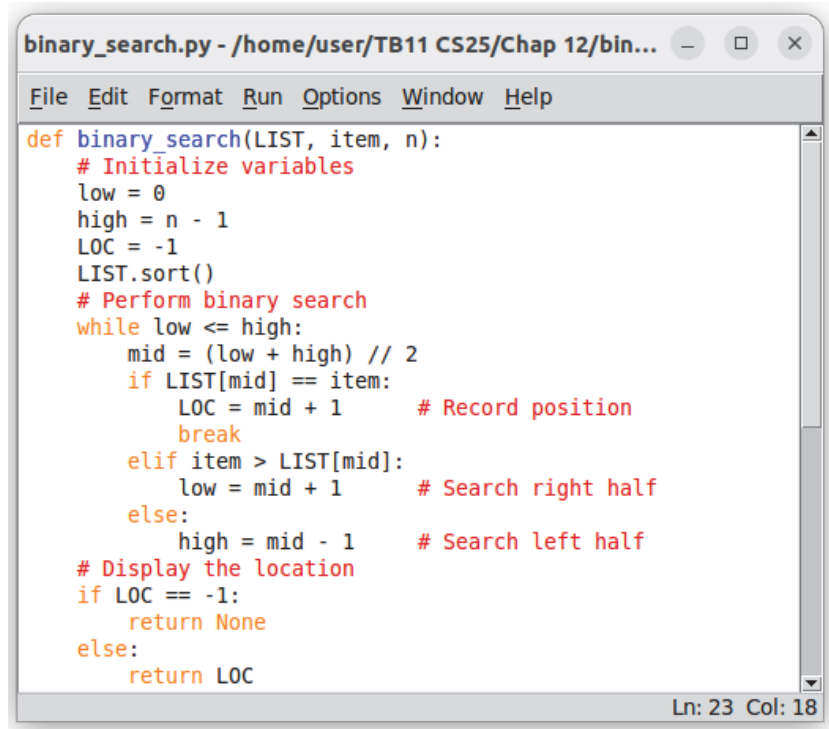
Algorithm for Binary search

- Step 1: Start.
- Step 2: Accept a value in N as the number of elements of the list.
- Step 3: Accept N elements into the list named LIST in sorted order.
- Step 4: Accept the value to be searched in the variable ITEM.
- Step 5: Set search location, $LOC = -1$.
- Step 6: Store the position of the first element of the list in FIRST and position of the last element in LAST.
- Step 7: Repeat Steps 8 to 12 while $(FIRST \leq LAST)$.
- Step 8: Find the middle position using the formula $(FIRST + LAST) / 2$ and store it in MIDDLE.
- Step 9: Compare the search value in ITEM with the element at the MIDDLE of the list.
- Step 10: If the MIDDLE element contains the search value in ITEM, then stop search, display the position and go to Step 13.
- Step 11: If the search value is larger than the MIDDLE element, then set $FIRST = MIDDLE + 1$.

- Step 12: If the search value is smaller than the MIDDLE element, then set LAST = MIDDLE - 1.
- Step 13: If the value at LOC is less than 0, then display "Not Found", else display the value of LOC + 1 as the position of the search value.
- Step 14: Stop.

The repeated operations of locating the required element or narrowing the list to half its size (steps 5 to 9) are grouped into a separate function. As with the previous algorithms, this approach improves clarity and keeps the program well organised.

The binary search function to search for an element in a list is shown in Figure 12.9 and the main function is shown in Figure 12.10. The output obtained is shown in Figure 12.11.

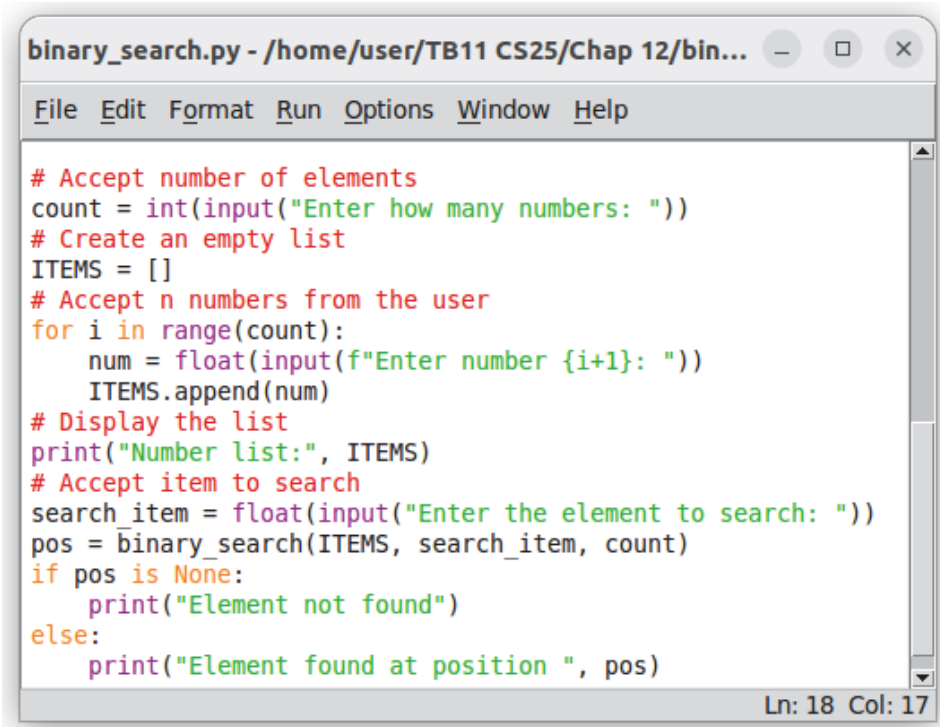


```
binary_search.py - /home/user/TB11 CS25/Chap 12/bin...
File Edit Format Run Options Window Help

def binary_search(LIST, item, n):
    # Initialize variables
    low = 0
    high = n - 1
    LOC = -1
    LIST.sort()
    # Perform binary search
    while low <= high:
        mid = (low + high) // 2
        if LIST[mid] == item:
            LOC = mid + 1      # Record position
            break
        elif item > LIST[mid]:
            low = mid + 1      # Search right half
        else:
            high = mid - 1     # Search left half
    # Display the location
    if LOC == -1:
        return None
    else:
        return LOC

Ln: 23 Col: 18
```

Figure 12.9: binary_search() function

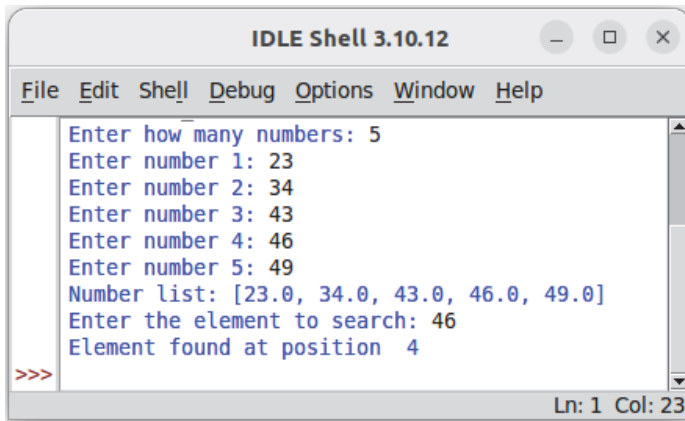


```
binary_search.py - /home/user/TB11 CS25/Chap 12/bin...
File Edit Format Run Options Window Help

# Accept number of elements
count = int(input("Enter how many numbers: "))
# Create an empty list
ITEMS = []
# Accept n numbers from the user
for i in range(count):
    num = float(input(f"Enter number {i+1}: "))
    ITEMS.append(num)
# Display the list
print("Number list:", ITEMS)
# Accept item to search
search_item = float(input("Enter the element to search: "))
pos = binary_search(ITEMS, search_item, count)
if pos is None:
    print("Element not found")
else:
    print("Element found at position ", pos)

Ln: 18 Col: 17
```

Figure 12.10: Main function that calls binary_search()

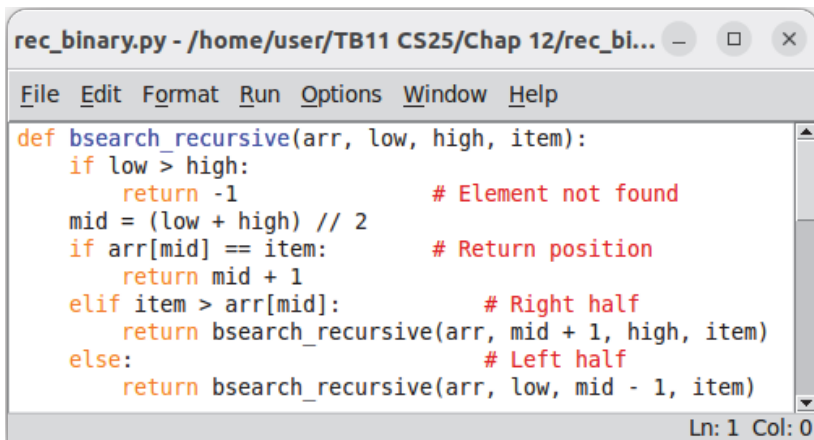


```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Enter how many numbers: 5
Enter number 1: 23
Enter number 2: 34
Enter number 3: 43
Enter number 4: 46
Enter number 5: 49
Number list: [23.0, 34.0, 43.0, 46.0, 49.0]
Enter the element to search: 46
Element found at position 4
>>>
Ln: 1 Col: 23

```

Figure 12.11: Output for binary search



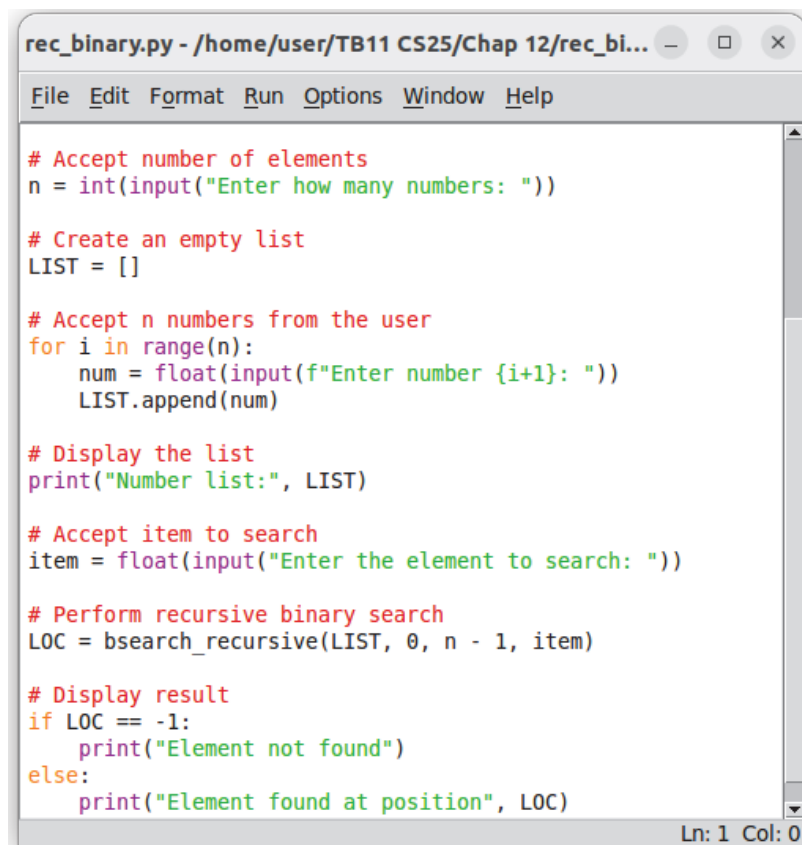
```

rec_binary.py - /home/user/TB11 CS25/Chap 12/rec_bi...
File Edit Format Run Options Window Help
def bsearch_recursive(arr, low, high, item):
    if low > high:
        return -1          # Element not found
    mid = (low + high) // 2
    if arr[mid] == item:    # Return position
        return mid + 1
    elif item > arr[mid]:   # Right half
        return bsearch_recursive(arr, mid + 1, high, item)
    else:                  # Left half
        return bsearch_recursive(arr, low, mid - 1, item)
Ln: 1 Col: 0

```

Figure 12.12: Recursive function for binary search
bsearch_recursive()

In Chapter 7, we learned that recursion is particularly useful for solving problems that repeat in smaller, similar patterns. Binary search is a good example of such a problem. In this technique, the algorithm begins by examining the middle element of the list. If this element matches the search value, the search is complete. If the search value is smaller, the algorithm recursively searches the first half of the list; if it is larger, the search continues recursively in the second half. Figure 12.12 explains how binary search can be implemented using a recursive function named **bsearch_recursive()** and Figure 12.13 gives the main function that calls the recursive function **bsearch_recursive()**.



```

rec_binary.py - /home/user/TB11 CS25/Chap 12/rec_bi...
File Edit Format Run Options Window Help

# Accept number of elements
n = int(input("Enter how many numbers: "))

# Create an empty list
LIST = []

# Accept n numbers from the user
for i in range(n):
    num = float(input(f"Enter number {i+1}: "))
    LIST.append(num)

# Display the list
print("Number list:", LIST)

# Accept item to search
item = float(input("Enter the element to search: "))

# Perform recursive binary search
LOC = bsearch_recursive(LIST, 0, n - 1, item)

# Display result
if LOC == -1:
    print("Element not found")
else:
    print("Element found at position", LOC)
Ln: 1 Col: 0

```

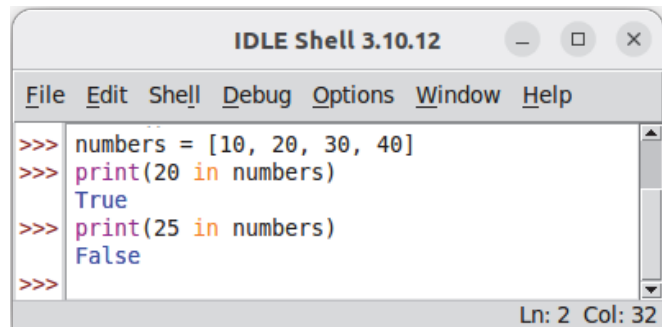
Figure 12.13: Main function that calls bsearch_recursive()

12.2.2 Searching Using Built-in Functions

Searching in a list can be performed in Python using **in** operator and using built-in function **index()**. Let us discuss them.

(a) **in** Operator

in operator checks whether a value exists in a list, string, or tuple. If found, it returns True and otherwise False. The use of **in** operator is shown in Figure 12.14.



```

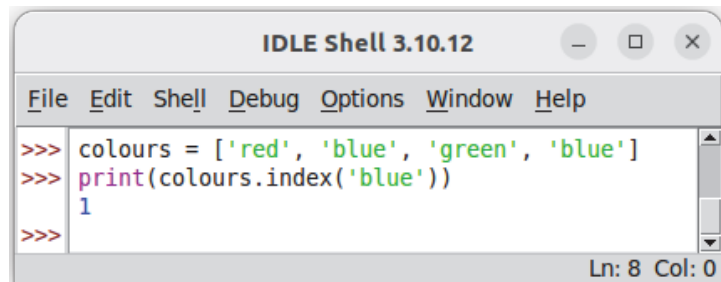
IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> numbers = [10, 20, 30, 40]
>>> print(20 in numbers)
True
>>> print(25 in numbers)
False
>>>
Ln: 2 Col: 32

```

Figure 12.14: Use of **in** operator for searching

(b) **index()** Function

index() function returns the index of the first occurrence of an element in a list. See Figure 12.15. If the element is not in the list, it raises an exception. We will discuss how to handle exceptions in Class 12. Note that index starts from 0.



```

IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
>>> colours = ['red', 'blue', 'green', 'blue']
>>> print(colours.index('blue'))
1
>>>
Ln: 8 Col: 0

```

Figure 12.15: Use of **index()** function for searching



Let us do

1. Bubble sort arranges elements by repeatedly swapping those that are out of order. After the first pass, the largest element moves to its correct position. In the second pass, the algorithm works on the remaining elements, and this process continues until the second last element. The bubble function shown below performs the necessary exchanges during each pass:

```

def bubble(LIST, last):
    for j in range(0, last):
        if LIST[j] > LIST[j + 1]:
            LIST[j], LIST[j + 1] = LIST[j + 1], LIST[j]
    return LIST

```

Write a Python function that uses this swap function to carry out the complete bubble sort and then run the program on a computer to verify its working.

2. Selection sort can also be separated into two functions. The inner function finds the smallest element in the unsorted section of the list and swaps it with the element at the first position. The outer function then calls this inner function $n-1$ times to finish sorting the entire list. Given that a function named `exchange(LIST, first, n)` returns the updated LIST, write complete function definitions for both the inner and the outer functions and then run the program on a computer to verify its working.

Sorting and searching are essential for organising data efficiently, and this chapter has explained both fundamental algorithms and Python's built-in techniques used to carry out these operations. By learning bubble sort, selection sort and binary search, students gained an understanding of how data can be arranged and accessed efficiently. Although Python provides convenient functions like `sort()`, `sorted()`, `index()`, and `in` operator, knowing how the underlying algorithms work enhances analytical thinking and programming confidence. These techniques form the groundwork for many advanced operations in computer science. With a solid grasp of sorting and searching, students become better prepared to design faster, more reliable programs and explore more complex algorithmic concepts.

Computers and Society

Computers have become an integral part of modern society, shaping the way we learn, work and communicate. As technology continues to advance rapidly, it brings with it new responsibilities and challenges. This chapter examines the interaction between computers and society by addressing key areas such as cyber ethics and social media etiquette, which promote responsible online behaviour and intellectual property rights that safeguard digital creations. It also introduces assistive technology, highlighting how computers can support and empower individuals with disabilities. In addition, the chapter discusses the free and open source software movement, which fosters collaboration, sharing and the open exchange of knowledge. Environmental concerns, including e-waste and green computing, are also explored, emphasising the need for environmentally responsible use of technology. Gaining an understanding of these topics enables individuals to use technology ethically, responsibly and for the greater good of the society.

13.1 Cyberspace

Cyberspace is the virtual environment created by interconnected computer networks, mainly the Internet. It enables people to communicate, share information, learn, work, and conduct business without physical boundaries. Activities such as sending emails, using social media, online banking, digital learning, and e-commerce take place in cyberspace. While cyberspace provides speed, convenience and global connectivity, it also poses challenges like cybercrime, privacy issues and misuse of information. Therefore, responsible behaviour, ethical use of technology and awareness of cyber safety are essential for maintaining a secure cyberspace. Cyber ethics guides users to behave responsibly and morally in the cyberspace.

13.1.1 Cyber Ethics

Internet has brought tremendous benefits in fields such as communication, business, education and entertainment. It has also raised crucial ethical issues. Cyber ethics refers to moral issues and decisions for responsible use of online technology. It involves understanding the ethical implications



of activities like data privacy, online security, digital communication and the responsible use of technology. Just like in the physical world, actions in the online world have consequence and understanding these consequences is crucial for anyone who uses digital technology.

The key concepts of cyber ethics are privacy, cyber security, digital footprint and online behaviour, and digital divide.

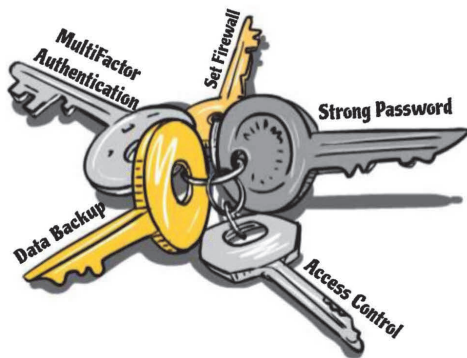
a. Privacy

Privacy is one of the most important concepts in cyber ethics, as it involves an individual's right to control personal information in the digital world. Today, many online services collect personal data such as our name, location, browsing habits and other sensitive details, sometimes without our clear permission or full awareness. Ethical concerns arise when this information is misused, shared without consent or exposed due to weak security. Protecting privacy means ensuring that personal data is collected transparently, stored safely and used only for the right purpose.

As responsible digital citizens, we must be aware of what information we share online and understand our rights to keep our data safe and private. Respecting others' privacy helps build trust and creates a safer online environment for everyone.

b. Cyber Security

Cyber security is an essential part of cyber ethics because it focuses on protecting computers, networks, and personal information from unauthorised access, misuse of data or attacks. As we rely more on digital tools for learning, communication and creativity, strong cyber



security practices ensure that data remains safe and online spaces remain trustworthy. Ethical cyber security means using technology responsibly - creating strong passwords, avoiding harmful activities like hacking and respecting the security of others' devices and information. We should work together to build a safe digital community by reporting suspicious activities and supporting others who may face online threats.

c. Digital Footprint and Online Behaviour

A digital footprint is the trail of information that a person leaves behind while using the internet, such as browsing history, social media posts, comments and data collected by websites or mobile applications. Data may be shared online knowingly or collected without their direct awareness. Since digital footprints shape a person's online reputation and can be permanent, it is important to manage them responsibly by thinking before posting, protecting personal information, and managing privacy settings.

Online behaviour refers to how individuals act on digital platforms. Practising good online behaviour includes using polite language, avoiding cyberbullying, respecting others' opinions, and not engaging in activities like hacking or plagiarism. Plagiarism is stealing someone else's work, ideas or words and presenting them as your own without giving proper credit to the creator. To protect our digital footprint and ensure responsible online behaviour, we should think carefully before sharing content online, protect personal information, use strong passwords, adjust privacy settings on social media, verify information before sharing, avoid suspicious links, behave respectfully towards others, log out from shared devices, and regularly review our online activities to maintain a safe and positive online presence.



d. Digital Divide

Digital divide refers to the gap between those who have access to digital technologies and those without. This raises significant cyber ethics concerns related to equity, privacy, and responsible technology. This divide stems from disparities in internet access, devices, and digital skills, often linked to economic, geographic or social factors. The digital divide increases vulnerabilities like data breaches for underprivileged users who lack secure tools. Students with online access may enhance their academic skills and adults may get more opportunities as most of the job applications are now online. It also increases inequality, as poor areas remain stuck without the skills needed for modern work, access to healthcare information, or the ability to stay connected with family. To reduce and protect society from the digital divide, efforts should be made to ensure equal access to digital devices and the internet, promote digital literacy through education and training, provide affordable technology and connectivity, support community centers and public access facilities.

13.1.2 Social Media Etiquette

Within cyberspace, social media platforms enable people to create, share, and exchange information instantly, transforming the way individuals communicate and interact. Social media etiquette, a key aspect of cyber ethics, requires us to follow principled guidelines for respectful, responsible online interactions that protect privacy, promote fairness, and prevent harm.

Think before posting: Avoid posting offensive, misleading, or fake content. Use polite language, and fact-check to stop misinformation from spreading.

Ensure privacy and mutual respect: Never share personal data, tag without permission, or post compromising photos/videos. Give credit for others' work and respect diverse opinions without judgment or cyberbullying.

Avoid harmful actions: Steer clear of spamming, memes, and stalking. Report abusive content and do not dominate discussions.

Follow platform-specific rules: Maintain proper digital behaviour by using clear subject lines in e-mails, avoiding loud media playback in public places, muting your microphone during online meetings while not speaking etc.

Build accountability: Update privacy settings, install security software, and treat online spaces like real life to foster digital citizenship and ensure legal compliance.



Let us do

Answer the following questions.

1. Define cyber ethics. Why is it important in today's digital world?
2. Explain the concept of privacy in cyberspace and mention one ethical concern related to it.
3. Define digital footprint. Why should digital footprints be managed carefully?
4. What is plagiarism? Why is it considered unethical in cyberspace?
5. Explain the term digital divide. Mention one effect of the digital divide on education or employment.
6. List any two rules that promote responsible use of social media.

13.2 Intellectual Property Rights (IPR)

Intellectual Property Rights (IPR) protect creations of the human mind, such as inventions, literary works, designs and symbols. They grant creators the exclusive rights to use, sell, or license their creations. It is normally given for a limited period of time. These rights encourage innovation by rewarding creativity while balancing public access. Intellectual Property Rights (IPR) in the IT field safeguard software, algorithms, databases, websites and digital innovations, preventing unauthorised copying. These protections encourage IT developers to invest in research and development while addressing cyber ethics by promoting fair use and discouraging piracy. In India, IPR is classified into patents, copyrights, trademarks, industrial designs, geographical indications and semiconductor integrated circuits layout design.

a. Patents: A patent is the exclusive right granted for an invention. It is the legal right obtained by an inventor for exclusive control over

the production and sale of the mechanical or scientific invention for a limited period of time. The term of every patent in India is 20 years from the date of filing the patent application. Once a patent expires, the protection ends, and the invention can be used by the public freely.

- b. Copyright:** Copyright is a legal right given to the creators for an original work usually for a limited period of time. Copyright applies to a wide range of creative intellectual or artistic forms of works which include books, music, painting, sculpture, films, advertisement and computer software. This covers rights for reproduction, communication to the public, adaptation and translation of the work.

Intellectual Property	
Registered trademark	®
Unregistered trademark	™
Copyright	©
Sound-recording copyright	!

Copyright can also be registered to give a legal status to a creative work. Piracy refers to the illegal copying, distribution, or use of copyrighted material without the permission of the copyright owner. This includes activities such as downloading or sharing movies, music, software, books, or videos without a valid license. Under Indian Copyright Act, a work is automatically protected by copyright when it is created. The general rule is that copyright lasts for 60 years after the death of the last surviving author.

- c. Trademark:** A trademark is a unique word, logo, symbol, name, slogan, shape, colour combination, or design that identifies goods or services from one business and sets them apart from others. A trademark must be registered. The initial term of registration is for 10 years after which it can be renewed. It protects brand reputation, stops consumer confusion, and fights counterfeits like fake products.
- d. Industrial design:** An industrial design refers to the ornamental aesthetic aspects of an article. A design may consist of features such as shape, surface of an article, patterns, lines or colour. It can be the stylish outer shell of mobile phones, the shape of soft drink bottles or the exterior parts of vehicles that make them attractive and unique. In India industrial designs protection lasts for an initial period of 10 years from the registration date which can be extended for a further 5 years.
- e. Geographical Indications (GI):** Geographical Indications are goods that have a specific geographical origin and possess qualities or a reputation that are attributable to that place of origin. Aranmula Kannadi, the traditional metal mirror made in Aranmula, Kerala, is a Geographical Indication (GI) product. It is handcrafted using a unique metal-alloy technique, giving it a mirror surface that is unlike ordinary glass mirrors. Only mirrors



made in Aranmula can be called 'Aranmula Kannadi,' protecting the craft from imitation and preserving this cultural heritage. Protection lasts indefinitely if renewed, preventing unfair practices in digital and physical markets. Palakkadan Matta rice, Kuthampully Silk, Vazhakulam pineapple etc. are some of the popular products with geographical indication from Kerala.

- f. **Semiconductor integrated circuits layout design:** It refers to the three-dimensional arrangement of elements such as transistors, resistors, and interconnections in an integrated circuit. It grants legal rights to creators of original and distinctive layout designs, preventing others from copying, reproducing or commercially using the design without permission. Such protection encourages innovation in the semiconductor and electronics industry by safeguarding the intellectual effort and creativity of designers. In India, the protection period is 10 years.

13.3 E-Waste

With the increasing use of computers, digital devices and technologies, the amount of electronic waste (e-waste) generated has also risen.

E-waste or electronic waste, refers to electrical and electronic equipment that has reached the end of its useful life. This includes items such as computers, laptops, smartphones, televisions, refrigerators, washing machines, batteries, circuit boards, printers and chargers.

E-waste is one of the fastest-growing types of waste in the world because technology is changing rapidly, and people frequently replace old devices with newer models. Improper disposal of e-waste is dangerous because it contains hazardous substances like lead, mercury, cadmium, brominated flame retardants and arsenic. These toxic chemicals can leak into the soil and groundwater or release harmful fumes when burned, leading to serious environmental pollution and health risks such as respiratory problems, skin diseases and nervous system disorders. The disposal of these devices poses



significant environmental challenges, necessitating practices like green computing and responsible e-waste management. The excessive use of AI increases the demand for powerful data centers, high performance computers and frequent hardware updates. This leads to higher electricity consumption, which increases carbon emission and contributes to environmental pollution.

Table 13.1: Hazardous chemicals in e-Waste and its impact

Hazardous Chemical	Source in e-Waste	Diseases / Health Effects Caused
Lead (Pb)	CRT monitors, batteries, solder in circuit boards	Damage to nervous system, kidney problems, learning disabilities
Mercury (Hg)	Fluorescent lamps, switches, LCD screens	Brain damage, memory loss, vision and hearing impairment
Cadmium (Cd)	Rechargeable batteries, semiconductors	Lung damage, kidney failure, bone weakness
Chromium (Cr VI)	Metal coatings, corrosion protector	Skin allergies, respiratory problems, cancer
Brominated Flame Retardants (BFRs)	Plastic casings, circuit boards	Hormonal imbalance, liver damage, developmental disorders
Beryllium (Be)	Motherboards, connectors	Lung disease, skin irritation, cancer
Arsenic (As)	Semiconductors, microchips	Skin diseases, cancer, digestive disorders

13.3.1 How e-waste can be reduced?

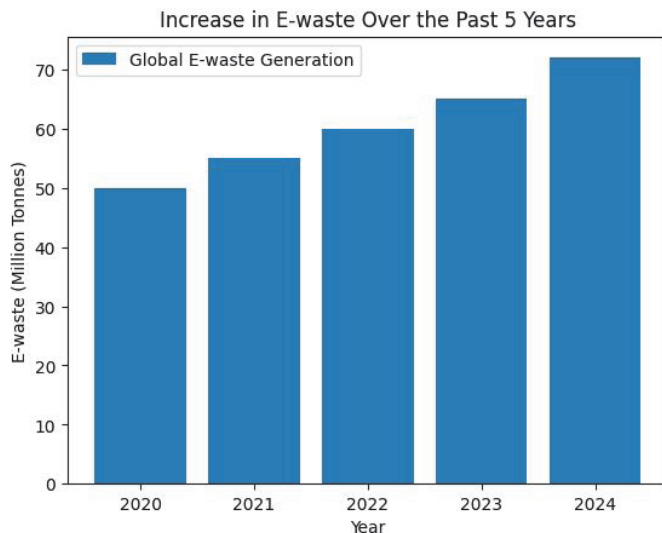
E-waste can be effectively reduced by making thoughtful and responsible choices while using electronic devices.

1. **Reuse and repair:** Use electronic items for a longer time by repairing them instead of replacing them.
2. **Donate or sell old devices:** Working devices can be donated to schools, charity or sold instead of being thrown away.
3. **Practise proper recycling:** Dispose electronic waste at authorised e-waste recycling centres so that harmful materials are handled safely.
4. **Buy eco-friendly products:** Choose products that are durable, energy-efficient, and designed to be recycled.
5. **Avoid frequent upgrades:** Do not replace gadgets unless necessary. Avoid buying new models just because they are trendy.

13.3.2 E-waste Disposal Methods

E-waste disposal methods focus on managing discarded electronic items in an environmentally responsible manner. Common methods include reuse, recycling, incineration and safe land filling to minimise environmental pollution and health risks. Figure 13.1 shows the global growth of e-waste over the past five years.

- a. **Reuse:** It refers to second-hand use or usage after the equipment has been upgraded or modified.



Source: Global E-waste Statistics Partnerships (GESP), 2024

Figure 13.1: Growth of e-waste

Reusing e-waste helps to extend the life of electronic products, reduces the demand for new devices, and conserves natural resources. Promoting reuse through donation, resale or refurbishment is an effective and eco-friendly way to manage e-waste responsibly.

- b. **Incineration:** It is a controlled and complete combustion process in which the e-waste is burned in specially designed incinerators at a high temperature in the range of 900 to 1000 degree Celsius.
- c. **Recycling of e-Waste:** Recycling is the process of making or manufacturing new products from a product that has originally served its purpose. Monitors, keyboards, laptops, modems, hard drives, mobiles, printers, CPUs, memory chips and cables can be recycled.
- d. **Safe land-filling:** Safe land-filling is a controlled method of disposing of waste, including certain types of e-waste, in specially designed landfill sites. It is used only for waste that cannot be recycled or reused, and it helps reduce environmental pollution and protect human health.



Figure 13.2: Green Computing Practices

13.4 Green Computing

In today's world, computers and digital devices are widely used in education, communication, business, entertainment and everyday life. But behind this comfort, technology also brings environmental challenges. Devices consume electricity and when discarded, create harmful electronic waste. Therefore, the idea of Green Computing encourages us to use technology in a way that is safe, efficient and friendly to nature.

Green Computing is the practice of reducing the environmental impact of computers and related technologies.

It focuses on saving energy, reducing pollution and managing e-waste properly. Figure 13.2 illustrates various green computing practices.

By supporting Green Computing, we help conserve natural resources, reduce pollution and slow down environmental damage. So, let us lead the society by initiating the following simple actions.

- Switch off devices when they are not in use.
- Use power-saving features like optimised power saving technique in gadgets.
- Encourage paper-saving by using e-billing, electronic documents and cloud storage.
- Recycle old gadgets through authorised e-waste collection centres.
- Choose energy-efficient devices such as LED lighting, energy star certified appliances, etc.

Let us make sure that our progress in computing also supports the well-being of our planet. Green Computing encourages us to be responsible users today, so that we can enjoy a cleaner environment in future.



Let us do

Answer the following questions

1.in e-waste from CRT monitors can cause damage to the nervous system.
2. Describe two methods to reduce e-waste generation in daily life.
3. State two practices to promote green computing in schools or homes.
4. Explain why high energy consumption by computers motivates the need for green computing.
5. Discuss the role of green computing in addressing e-waste and natural resource exhaustion.

Perform the following activities

- Identify e-waste around you.
- Make a list of such items found in your home, school and surroundings. Conduct an awareness class about the impacts of e-waste in our environment. Conduct campaigns for reduction and safe disposal of e-waste.

13.5 Free and Open Source Software (FOSS) Movement

The free and open source software movement emerged primarily as a response to the challenges of proprietary software, such as

Proprietary software is owned and controlled by an individual or a company. The source code is not made available to users, and they are not allowed to modify, copy, or distribute the software without permission. Users can only use the software according to the terms and conditions of the license. MS Office, Adobe Photoshop, Norton Antivirus are some proprietary software.

restrictive licensing, lack of transparency, and vendor lock. Free and open source movement proposed the idea that software source code should be freely available for anyone to view, use, modify and distribute.

Examples of free and open source software include Ubuntu, Mozilla Firefox, LibreOffice, GIMP, VLC Media Player, Python, MariaDB etc. These software allow users to use, study, modify, and distribute the source code freely.

The free software foundation defines the four freedoms for free and open source software. They are,

Freedom 0: The freedom to run the program for any purpose.

Freedom 1: The freedom to study how the program works and adapt it to your needs. Access to source code is provided.

Freedom 2: The freedom to distribute copies of the software.

Freedom 3: The freedom to improve the program and release your improvements to the public, so that the whole community benefits.

Richard Stallman, who founded the GNU project and the Free Software Foundation and Linus Torvalds, who created the Linux kernel, were instrumental in pioneering the core tenets of software freedom and collaborative development that defined the open source movement. Other significant personalities in FOSS include Eric S. Raymond and Bruce Perens, who co-founded the Open Source Initiative (OSI) to promote open source to the business world and technologist Christine Peterson, who is credited with coining the term "open source" itself.



Let us do

Answer the following questions

1. is a key figure who created the Linux kernel.
2. Name three examples of Free and Open Source Software mentioned.
3. Discuss how the FOSS movement supports cyber ethics and innovation.
4. List and explain four freedoms defined by free software foundation.

13.6 Assistive Technology

Assistive Technology refers to the, devices, software and computer systems designed to support individuals with disabilities or special needs. In today's digital society, where education, employment, governance and communication are deeply dependent on computing, inclusiveness is not optional but essential. If technology is designed without considering diverse users, it can unintentionally exclude people with visual, hearing, motor or cognitive impairments, thereby widening the digital divide.

People with conditions such as visual impairment, hearing loss, speech difficulties, locomotor disabilities or neurological disorders like Amyotrophic Lateral Sclerosis (ALS) often face challenges in using conventional input and output devices.

Assistive technologies help overcome these barriers by enabling users to interact with digital systems independently, safely and with dignity, ensuring equal participation in learning, work and social life.

Assistive technology includes both hardware and software solutions, many of which are now built into modern operating systems and smart devices. Screen readers, screen magnifiers and braille displays support visually challenged people. Captioning tools, descriptive video services and visual alerts highly support people with hearing problems. Text to speech software, alternative input devices and on-screen keyboards are helpful for people with speech issues. Real time translators, transcribers and emerging artificial intelligent tools also assist us to a big extend.

13.6.1 Importance of Assistive Technology

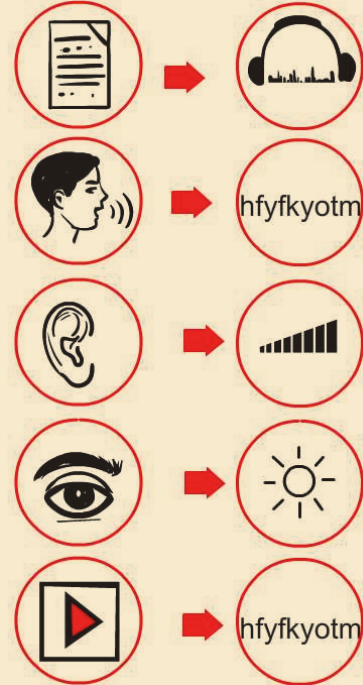
Assistive Technology plays a vital role in promoting digital equality, reducing dependency on others, ensuring ethical and responsible use of technology, and empowering individuals with disabilities to participate fully in society.



Imagine controlling a computer just by thinking.

Brain-Computer Interface (BCI) is an emerging Assistive Technology that enables direct communication between the human brain and a computer without using hands or speech. It is especially helpful for people with severe motor or speech disabilities, such as paralysis or neurological disorders.

BCI is not a fancy or imaginary idea—it is an active field of research worldwide. Many scientists, engineers, and medical researchers are working on BCI to make technology more inclusive and accessible. Students are encouraged to search about BCI and explore how computing, artificial intelligence, and neuroscience work together to improve human life.





With growing awareness of accessibility, many assistive features are now integrated into operating systems, smartphones, and web platforms, reinforcing the idea that accessibility is a fundamental principle of modern computing.



Let us do

Answer the following questions

1. What is assistive technology?
2. Explain why assistive technology is important.
3. How does assistive technology promote inclusiveness?

Computers have become an integral part of modern society, influencing the way we communicate, work, and solve problems. While technological advancements bring numerous benefits, they also come with responsibilities, such as ensuring cyber security, respecting cyber ethics and protecting intellectual property rights. Sustainable practices like green computing and proper e-waste management help minimise environmental impact, while assistive technologies ensure inclusivity for all. Additionally, movements like free and open source software promote collaboration and accessibility. Understanding these aspects equips individuals to use technology responsibly and ethically, contributing to a safer, more equitable and sustainable digital world.