

INTRODUCTION TO COMPUTING

A COURSE MATERIAL FOR

CSC 102

S. O. AKINOLA

**Associate Professor,
Computer Science Department,
University of Ibadan**

General Introduction and Course Objective

General Introduction

Computer Science is all about science of computing. It bothers on creation and use of computational tools such as hardware and software. In this course, students shall be introduced to the profession: Computer Science, hardware and software evolutions, character representations in computers, problems solving techniques in computing and a gentle introduction to computer programming using Python as a programming language tool.

Course Curriculum Contents

The definitions and scope of computer science, General structure of a computer system. Historical Development of computer systems, Generations of Computer system, ,Fields and Interests in computer science, Information representation in a computer, Basic Computer Organization, Instruction sets, Problem Solving using pseudo-codes, Programming Languages, Computer Networks and the Internet, Computers and Society. Introduction to Computer Programming using a suitable programming language.

Semester 1, LH 45; PH 45; 4U; Status C

Table of Contents

General Introduction and Course Objective

Study Session 1: Introduction to Computer Science

Study Session 2: Introduction to Computers

Study Session 3: Other Hardware Components of Computers

Study Session 4: Software Components of Computers

Study Session 5: Generations of Hardware and Software

Study Session 6: Introduction to Computer Networks and Internet

Study Session 7: Data and Information Representation

Study Session 8: Problem Solving with Computers

Study Session 9: Elements of Computer Programming I

Study Session 10: Elements of Computer Programming II

Study Session 11: Programming with Python Language

Study Session 12: Python Basic Operations

Study Session 13: Control Structures in Python Language I: Making Decisions

Study Session 14: Control Structures in Python Language II: Loops in Python

Study Session 15: Functions in Python Language

Study Session 16: Composite Types

Study Session 1: Introduction to Computer Science

Expected Duration: 1 week or 2 contact hours

Introduction

In this Session, you will be introduced to the Computer Science profession. The meaning, philosophy, areas of study as well as the relationship of Computer Science with other disciplines will be explained.

Learning Outcomes

When you have studied this session, you should be able to explain:

- 1.1 The meaning of Computer Science
- 1.2 Philosophy of Computer Science Profession
- 1.3 Areas of Study in Computer Science
- 1.4 Qualities of a Good Computer Scientist
- 1.5 Careers in Computer Science
- 1.6 Computer Science in Relation with Other Disciplines
- 1.7 Performance Evaluation of Computational Tools

1.2 What is Computer Science?

Computer Science originated in the mid-1940s with the joining together of algorithm theory, mathematical logic and the invention of the stored-program electronic computer (Denning, 1989). Although David Hilbert, Kurt Godel and Alan Turing laid the theoretical foundations of the discipline in the 1930s, the dearth of human ‘computers’ to calculate ‘firing tables’ for new missiles during World War II led to massive efforts to build the first modern Computer in the 1940s: Harvard Mark I and Electronic Numeric Integrator and Calculator (ENIAC).

If X number of Computer Scientists were to define Computer Science, we will get X number of definitions; each reflecting individual respondent’s perception of the discipline. Mathematicians claim it is a branch of mathematics, dealing with solving computational problems in Mathematics. Some people see it as the study of developing computer algorithms and programs. According to the Anthony Ratson’s Encyclopedia of Computer Science, Computer Science is concerned with information processes, with the information structures and procedures that enter into representations of such processes and with their implementation in information processing systems. According to Osofisan and Akinkunmi (2012), Computer Science is about building computers and writing computer programs, how computational tools are designed, developed and utilized.

In a nutshell, the author wishes to define Computer Science as the study of science, technology and art of design, implementation and utilization of computational tools such as the hardware and software components of computers. The science aspect of Computer Science deals with providing a formal basis for its ideas while its technology deals with providing tools and techniques for practical utilization of its ideas. The art aspect is inferred from its technology. Creativity and logical reasoning are very much needed in designing the computational tools (software and hardware).

Computer Science is an applied science with ideas built around the use of computer machine as an information processing device. Unlike such natural sciences like Biology, Chemistry and Physics that deal with natural phenomena, Computer Science mostly deals with man-made phenomena – models of real world entities and processes. Hence, its concepts and ideas are abstract in nature and intangible in essence. Solutions to real-world problems are presented in form of abstract models which are amenable to computer representation and manipulation. As computers require absolute precision in the formulation of such solutions to problems, such models are approximations of the real-world application system involved.

1.2 Philosophy of Computer Science Profession

Computer Science as a discipline is set out from its early days to produce experts that would create Information Technology Solutions to meet the needs of the Information Technology (IT) Industry. As such it has its vision set on peddling the creativity of Information technology tools, that is backed up with a deep and rigorous understanding of Computer Science theory as opposed to a routine and context specific creativity. Apart from solid theory being the key to good designs, a good understanding of Computer Science theory will also enhance adaptability across platforms and products.

1.3 Areas of Study in Computer Science

The conception, formulation, computer implementation, analysis and evaluation of procedures (algorithms) for a broad variety of problems constitute a major part of the computer Science activities. Closely associated with these activities are efforts to develop schemes, means and tools for building and executing procedures such as languages, major principles for structuring procedures, programming mechanisms, computer organizations and design aids to facilitate these efforts. In addition, a significant amount of effort is directed to the design of advanced systems – software and hardware. All these activities have important connections with several theoretical efforts in the field, some in application areas and others in the analysis of algorithms, formal languages, automata theory, switching theory and systems analysis.

An outline of major areas of study in Computer Science include but not exhaustive:

- (i) Representations in computer languages of problems, data and procedures in various applications – numerical and non-numerical.
- (ii) Theory of Computation and Analysis
- (iii) High level languages for various application areas, schemes for structuring data and procedures, language descriptions and translation schemes.
- (iv) Machine level languages, storage schemes and programming mechanisms
- (v) System organizations, executive and control mechanisms, computer system designs
- (vi) Theory of formal languages, automata theory and switching theory
- (vii) Data Mining, Big Data Analytics, Machine Learning, Data Science
- (viii) Computer Networking, Internet and Web Programming
- (ix) Wireless Computing
- (x) Software Engineering, Databases, Bioinformatics, Artificial Intelligence, etc

1.4 Qualities of a Good Computer Scientist

Qualities of a good Computer Scientist include:

- Passion for finding elegant solutions
- Ability to use mathematical analysis and logical rigour to evaluate such solutions
- Creativity in modelling complex problems through the use of abstractions
- Attention to details and hidden assumptions
- Ability to recognize variants of the same problem in different setting and
- Ability to retarget known efficient solutions to problems in new settings, i.e., reusability

1.5 Career Prospects in Computer Science

Many students graduate to rewarding computer-related careers in:

- Software engineering
- System Analyst
- Bioinformatics
- Data Scientists
- System administration and management
- Research and development in industrial and governmental laboratories. Etc.

1.6 Computer Science in Relation with Other Disciplines

Mathematics: Computer Science is widely considered a mathematical science. The reasons for this are not far-fetched. Computer Science and Mathematics have a common concern with formalisms, symbolic structures and their properties. Both put emphasis on general methods and problem-solving tools that can be used in great variety of situations. A subject as Numerical Analysis is studied in both disciplines. In fact a student of Computer Science is normally expected to have a sound mathematical background.

Engineering Science: Computer Science is also considered an engineering science. The structure of a computer system consists of physical components (the hardware) in the form of electronic or electromechanical building blocks for switching, storage and communication of information and programs (software) for managing the operation of the hardware. In the logical and system design of the computer, the designer is concerned with the choice of hardware and software building blocks and with their local and global organizations in the light of given operational goals for the overall systems. These design activities have strong points of contact with work in electrical engineering and in the emerging field of Software Engineering. They are also important subjects of study in Computer Science.

Every transition from the specification of an information processing task involves a design process. In many cases, these processes are highly complex and their effectiveness is strongly dependent on the availability of appropriate methodologies and techniques that may be used to guide and support them. This is one of the reasons why Computer Science is concerned with the methodologies of systems analysis and design. The concern is shared not only with engineering, but with other **decision-oriented disciplines** such as Business Administration and Institutional Planning. There is a more fundamental reason for a close coupling between Computer Science

and science of design. It comes from the concern of Computer Science with the information processes of problem-solving and goal-oriented decision-making, which are at the core of design. Processes of this type are objects of study in Artificial Intelligence (AI), a branch of Computer Science.

Computer Engineering: Computer engineering concerns itself with current practices in assembling hardware and software components to erect computing engines with the best cost-performance characteristics. In contrast, computer scientists worry about the feasibility and efficiency of solutions to problems in a manner that is less dependent on current technologies.

Several other disciplines are recognized as having domains of interest which overlap with Computer Science. **Library Science** which deals with the problems of organizing and managing knowledge, and of designing systems for its storage and retrieval (in the form of documents and facts), has this as a common concern with Computer Science. In fact, **Information Science** which embraces library science, concerns mainly with the processes of communication, storage, management and utilization of information in large database systems, has this domain falling into the broader domain of Computer Science.

Linguistics is another discipline whose domain of interest overlaps with Computer Science. The concern with language and communication is shared by the two disciplines. The study of linguistic processes and of related phenomena of “understanding”, establishes a special bond between Computer Science and **Psychology**. Psychological research in information processing models of cognition, perception and other mental functions has a substantial overlap with Computer Science. It is worth noting that a good deal of theories of formal languages come from linguistic theories, e.g., Chomsky’s theory of Language Hierarchy.

Philosophy: The study of certain theoretical questions about processes of reasoning by computer (performing deductions, forming hypotheses, using knowledge effectively in problem-solving processes) has created points of contact between certain parts of **Philosophy** (logic, epistemology, methodology) and Computer Science. The uses of these philosophy-based ideas are found in the Computer Science areas of databases and artificial intelligence, where logic-based languages have been developed (e.g. Prolog and Protégé). Also it is worth noting that digital computers are made up of various types of circuits, arranged and grouped according to the rules of Symbolic Logic.

There is a large “surface of contact” between Computer Science and other disciplines where new computer applications are being developed. Virtually all disciplines are involved in this contact. For instance, we have Computational Biology (Bioinformatics), Computational Chemistry, Computational Physics, etc. The nature of the contact is similar to the relationship between Mathematics and the physical sciences. This relationship involves the representation of scientific problems in mathematical systems wherein the problems can be studied and solved. In the case of Computer Science, the contact involves the representation of knowledge and problems of a discipline in forms that are acceptable to computers and the development of computer methods for the effective handling of these problems. Since computers can be made to represent and manipulate problems of enormous variety and complexity, it is likely that the extent of fruitful contact between Computer Science and other disciplines will be much larger than the

contact between Mathematics and the “mathematics utilizing” disciplines. In particular, it is likely that the role played by Computer Science in behavioural and Social Sciences and the humanities will be similar to that played by Mathematics in the growth of physical sciences.

Computers and software artifacts have become indispensable tools in every scientific discipline. The scientists who use Computer and software artifacts are called Computational Scientists. The use of computers has enabled:

- Biologists to comprehend genetics.
- Astrophysicists to get within femtoseconds (10^{15} sec) in laser technology, and
- Geologists to predict earthquakes.

Scientists in these disciplines increasingly rely on computational methodologies in addition to traditional mathematical or empirical methodologies to make advances in their respective fields of study

1.7 Performance Evaluation of Computational Tools

Computer Scientists build and study tools based on one or any combination of the following set of performance goals.

(i) Correctness

A tool is said to be correct if it works according to a pre-defined specification. The pre-defined specification must be clearly and unambiguously stated for correctness to be formally assessed.

(ii) Efficiency

The efficiency of a tool is a measure of its ability to make judicious use of resources. The most important resource for tools is time. Another important group of resources is machine resources (especially memory space). A tool is more efficient with respect to time if it uses less time to solve a problem than another tool. A tool is more efficient with respect to space if it uses less space to solve a problem than another. Time efficiency for software tools is normally measured as a mathematical function of the problem size. The smaller the function growth as problem size grows, the more time efficient the tool.

(iii) Security

A tool is secure if its functions will not be hampered by malicious attacks or other compromising situations such as access by unauthorised users and hackers. For example, hackers can interrupt information during transmission. Data interception can be prevented by data encryption. Data encryption is the process of transforming data into an unintelligible form (encrypted data) prior to transmission. The transformation is undone at the point of reception (decryption).

(iv) User-friendliness

User-friendliness reflects the ease with which a tool can be used by a human user. Factors ranging from culture and ergonomics to personal preferences may affect user-friendliness. Hence, user-friendliness may be context-dependent.

(v) Fault-tolerance

Fault-tolerance is the ability of a tool to cope in the presence of faults. A tool is fault-tolerant if every possible fault does not lead to a failure. The level of fault-tolerance is measured by *Mean Time Between Failure* (MTBF). This is a measure of the time between two failure incidents in the use of a tool.

(vi) Intelligence

Intelligence is the ability of a tool to mimic human intelligence. Tools can mimic intelligence in a number of ways including Reasoning and Language understanding.

Summary

The following concepts about Computer Science Profession have been introduced to you in this Session.

1. The Meaning Computer Science?
2. Philosophy of Computer Science Profession
3. Areas of Study in Computer Science
4. Qualities of a Good Computer Scientist
5. Careers in Computer Science
6. Computer Science in Relation with Other Disciplines
7. Performance Evaluation of Computational Tools

Self-Assessment Questions

1. Attempt to define Computer Science in your own words.
2. Why do you prefer to study Computer Science amidst other disciplines?
3. State three major works of a computer Scientist

Study Session 2: Introduction to Computers

Expected Duration: 1 week or 2 contact hours

Introduction

Computers are electronic machines that accept data, process it and output the results of the processing for users' utilization. In this Session, you will learn about the meaning of computers, general characteristics of computers and benefits of using computers. The different types of computers are not left out. .

Learning Outcomes

When you have studied this session, you should be able to explain:

- 2.1 The meaning of a Computer
- 2.2 Characteristics of a Computer System:
- 2.3 The Benefits of Computers to Organizations
- 2.4 Types and Classes of Computers
- 2.5 How Computers Work
- 2.6 The Components of a Computer System

2.1 What Is a Computer?

A computer is any electronic machine which can accept inputs (instructions and data) presented to it in a prescribed format from input devices, carry out some operations (process) on the input automatically, and supply the required results (output) in a specified format on output devices, as information for human decision-making, or as signals to control some other machines or process. This definition shows that Computers' job is to accept data as input, process the data and output the results (Input – Process – Output).

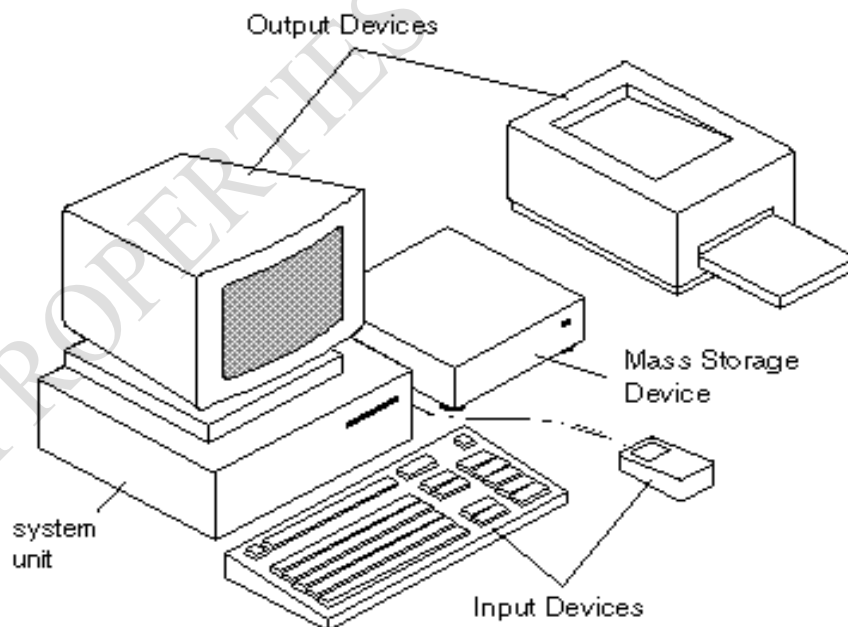


Figure 2.1: General Structure of a Computer System

Today computers do much more than simply compute: supermarket scanners calculate our bills while keeping store inventory, computerized telephone switching centres play traffic cop to millions of calls and keep lines of communication untangled; and automatic teller machines (ATM) let us conduct banking transactions from virtually anywhere in the world. Above all, the computer can store and manipulate large quantities of data at very high speed.

2.2 Characteristics of a Computer System:

Computers of all sizes have common characteristics, principally:

- (a) **Speed:** The Computer is made up of electronic components that operate at electronic speed – far more comparable to any other known processing information tool. They provide the processing speed required by all facets of society.
- (b) **Large storage Capacity:** With the use of external storage devices of large sizes and increasingly large memory capacity, huge amount of information can be stored, retrieved and processed electronically.
- (c) **Reliability:** The computer accuracy is consistently high when compared with other information processing tools. Computers are indeed extremely reliable.
- (d) **Automatic:** Once a program is stored in the computer's memory, the individual instructions will require less human intervention for the operation to be performed.
- (e) **Diligence:** A computer does not suffer from human trait of tiredness, as the computer is ready to perform any job as instructed.
- (f) **Versatility:** the fact that a computer can be used in any discipline has made it to be popular among other information processing tools. It is a general purpose device
- (g) The Computer is **Electronic** in nature and can process millions of instructions per second. They are productive.
- (h) The computer is not a **Substitute for Human Brain**.

2.3 The Benefits of Computers to Organizations

Unless a computer system can show a good value (profit) to the Organization, there can be no good reason for installing it. Unfortunately, it is not easy to quantify either benefits or costs absolutely and there are always mitigating circumstances which can invalidate powerful benefit or magnify cost beyond reason. However, outlined below are some of the benefits that Organization may expect to achieve.

(i) Dealing with increased volumes of transaction

A reasonably well-prepared data processing system should allow the clerical staff to deal with a significantly greater volume of work. Input via input devices (keyboard, mouse etc) is quicker than via most manual systems, the computer does the arithmetic (infallibly).

(ii) Better working conditions for staff

Computer can work rapidly, tirelessly, obediently and almost never make mistakes. It can take over boring and routine jobs, freeing staff to concentrate on other things'.

(iii) Better management of information

Using flexible software in the form of Database Management Systems (DBMS) enables management to look at stored information in files. Graphs can be plotted on revenue, costs, etc, and analyzed in different ways, shape or form.

(iv) Better control over costs

By identifying, through their DBMS, where costs are at variance with budgeted figures, action can be taken much more quickly than was ever possible with a manual system. The cost in question can range through stocks, overtime, overheads, postages, cash inflows and outflows, deposits and withdrawals etc.

(v) Faster results

Some companies have found that month-end figures can be produced at the end of the last day of the month once a computer system is functioning where previously it was about four weeks after end of month. Any routine operation, such as producing accounts can be done more quickly and more accurately by a computer than was ever possible manually. Cash position can be reported at a touch of a button after each transaction. Under the manual system, this is possible only hours after the end of the transaction day. More rapid and reliable results can also be obtained in the design office, technical services, engineering or any department that has to crunch numbers regularly. The use of special equipment such as plotters, and graphics terminals can also help in improving information production.

2.4 Types and Classes of Computers

Digital Computers are often classified according to their processing power, size and functions. The size of computers varies widely from tiny to huge and is usually dictated by computing requirements.

- ***Supercomputers***

They are the most powerful, the most expensive, the largest in size and the fastest. They are capable of processing trillions of instructions per second. They tend to be used primarily for scientific applications in weather forecasting, aircraft design, nuclear research, space research, and seismic analysis. Commercially, they are used as “host” processors and large networks that process data from thousands of remote stations. An example of a supercomputer is Cray-1 Supercomputer built by Cray Research Company. Supercomputers are used for tasks that require mammoth data manipulation.

- ***Mainframes***

Mainframe computers process data at very high rates of speed, measured in the millions of instructions per second. They are very expensive, large (often filling an entire room) and costing millions of dollars in some cases. Mainframes are designed for multiple users and process vast amounts of data quickly. Banks, insurance companies, manufacturers, mail-order companies, and airlines are typical users. Mainframes are often ‘servers’-- computers that control the networks of computers for large companies. Example of Mainframe computer is IBM 370 located at the University Computing Centre, U.I in the 1970’s.

- ***Mini-Computer***

This type of computer is often used by medium size business organizations for stock control and invoicing. The cost of mini-computer is lower than that of Main-Frame and generally suits the need of medium-sized business. Mini computers are small compare to main frame computer as the processor and peripherals are physically smaller. They possess most of the features found on mainframe computers, but on a more limited scale.

- ***Microcomputers***

These are the types of computers, which are generally used in colleges and institute for teaching purpose. They are terribly limited in what they can do when compared to the larger models discussed above because they can only be used by one person at a time, they are much slower than the larger computers, and they cannot store nearly as much information, but they are excellent when used in small businesses, homes, and school classrooms. These computers are comparatively inexpensive and easy to use.

Microcomputers can be divided into two groups -- personal computers and workstations. Workstations are specialized computers that approach the speed of mainframes. Often microcomputers are connected to networks of other computers. The price of a microcomputer varies greatly depending on the capacity and features of the computer. Microcomputers make up the vast majority of computers.

They are further categorized into: desktop, Laptop, Notebook, Palmtop and the smallest in size is the handheld computer called a ***personal digital assistant*** or a ***PDA***. PDAs are used to track appointments and shipments as well as names and addresses. PDAs are called pen-based computers because they utilize a pen-like stylus that accepts hand-written input directly on a touch-sensitive screen.

Personal Computers

To quote Wikipedia, "A personal computer (PC) is a computer whose original sales price, size, and capabilities make it useful for individuals." Personal computers are computers that are used by one person at a time. Most computers in existence today are personal computers. Personal computers are most commonly used at home or in an office, but they can be found elsewhere, especially if the personal computer in question is a laptop. The concept of a personal computer began in the 1970s with important individuals involved in academics or research owning a personal computer for use in the workplace. The computers were too expensive for most people to purchase, but all that changed in 1975 with the invention of the microprocessor. This allowed computers to be much smaller and much cheaper. The use of personal computers continued to grow throughout the 1980s with more development making computers cheaper, smaller, and easier to use. In the 1990s, computers became even more powerful causing them to be mainstream in many homes. Today, computers are better than ever and are used by people for work, games, information, and other things.

2.5 How Computers Work

Understanding *how* your computer works is a gateway to understanding what you can do with your computer. At the heart of any computer, modern or early, is a circuit called an ALU, or Arithmetic Logic Unit. It's comprised of a few simple operations which can be done very quickly. This, along with a small amount of memory running at processor speed called registers, make up what is known as the CPU, or Central Processing Unit.

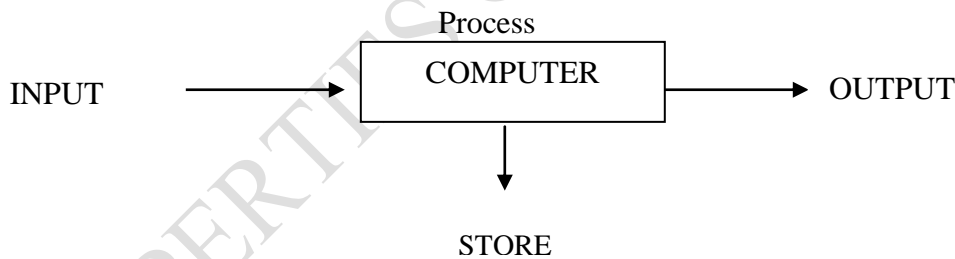
A CPU isn't very useful unless there is some way to communicate to it, and receive information back from it. This is usually known as a Bus. The Bus is the input/output, or I/O gateway for the CPU. The primary area with which the CPU communicates with its system memory in what is commonly known as RAM, or Random Access Memory. Depending on the platform, the CPU may communicate with other parts of the system, or it may communicate just through memory.

When a computer is asked to do a job, it handles the task in a very special way.

1. It accepts the information from the user. This is called input.
2. It stored the information until it is ready for use. The computer has memory chips, which are designed to hold information until it is needed.
3. It processes the information. The computer has an electronic brain called the Central Processing Unit, which is responsible for processing all data and instructions given to the computer.
4. It then returns the processed information to the user. This is called output.

Every computer has special parts to do each of the jobs listed above

A computer collects, processes, stores and outputs information as illustrated by the figure below:



- **INPUT**

Computer receives inputs (instructions and data) through input devices. Input device lets you communicate with a computer. You can use input devices to enter information and issue commands. Keyboard, mouse and joystick are input devices.

- **PROCESS**

The Central Processing Unit (CPU) is the main chip in a computer. The CPU processes instructions, performs calculations and manages the flow of information through a computer system. The CPU communicates with input, output and storage devices to perform tasks.

- STORE

Inputs and sometimes processing results are stored on storage devices. A storage device holds information. The computer uses information stored on these devices to perform tasks. Hard drives, tape drive, floppy disk, Flash Drive are storage devices.

- OUTPUT

Outputs from a computer are received through output devices. An output device lets a computer communicate with you. These devices display information on a screen, create printed copies or generate sound. A monitor, printer and speakers are output devices.

2.6 The Components of a Computer System

A system is made up of several components working together to achieve a common goal. The Computer system is divided into two broad subcomponents viz: HARDWARE and SOFTWARE.

$$\text{Computer system} = \text{hardware} + \text{Software}$$

2.6.1 The Hardware Subcomponent

The hardware consists of all the electromechanical gadgets that make up the computer system i.e. all the physical aspect of the computer (any part of a computer system you can see or touch). Some are external (e.g. the monitor, the mouse, the keyboard, the speaker etc.) while others are internal i.e. embedded in the computer casing (e.g. Processor, memory, motherboard, sound card, video card etc). Every Computer system, whether it is a multimillion Naira mainframe or thousands Naira personal computer, has the following four components, Input unit, Memory, Central Processing unit, and Output unit. The input, output and storage devices are also referred to as peripheral devices

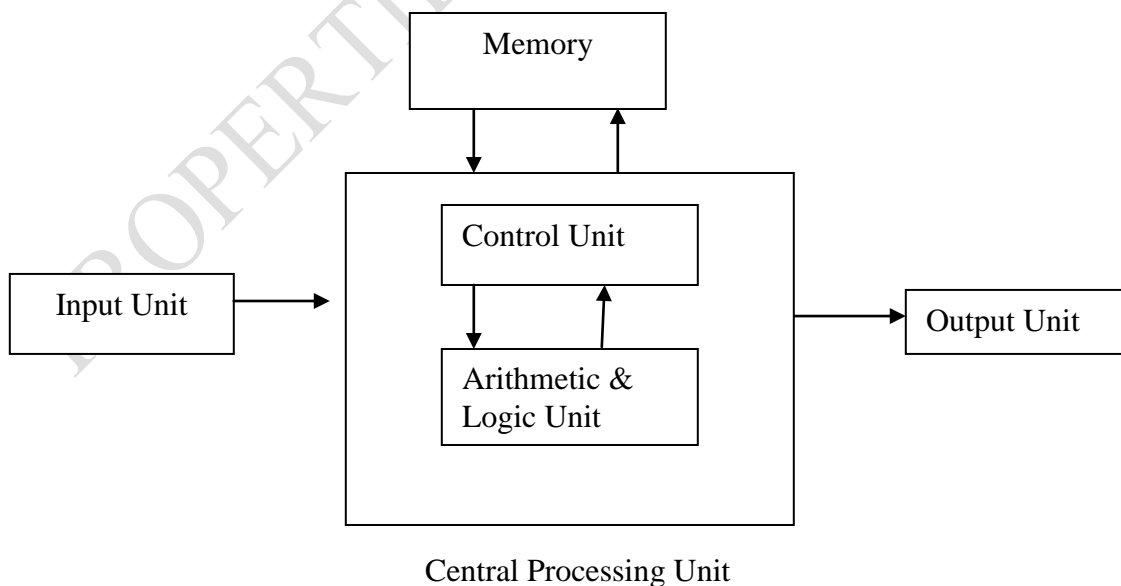


Figure 2.2 A simple Computer Model

- **Input Unit**



The Input Unit consists of devices through which information to be processed is passed on to the computer. Such devices transform source information to be processed from human readable form into internal form which is transmitted to the Central Processing Unit. One of the best features of a computer is the ability to give the computer commands and feed it information. Without an input device this would not be possible. Input devices can be built into the computer, like the keyboard in a laptop, or it can be connected to the computer by a cable. The most commonly used input devices are the **keyboard and mice**. There are lots of others such as: trackballs, touch pads, touch screens, pens, joysticks, scanners, barcode readers, video and digital cameras, web cameras and microphones.

- **Output Unit**



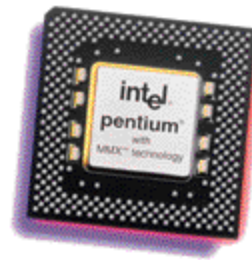
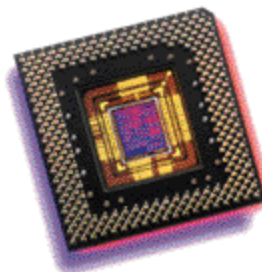
The output unit consists of devices that change the internal form of processed information (inform of electrical pulses) from the CPU into human readable or audible form. Common output devices include:

- Printers
- Monitor
- Speakers
- Headset
- Video output devices

- **The Central Processing Unit (CPU)**



Pentium® II processor



The Central Processing Unit often referred to as the **CPU** or **microprocessor** is the brain of the computer. The *CPU* consists of electronic circuits that interpret and execute instructions; it communicates with the input, output, and storage devices. The central processing unit (CPU) is the portion of a computer system that carries out the instructions of a computer program, to perform the basic arithmetical, logical, and input/output operations of the system. The CPU plays a role somewhat analogous to the brain in the computer. The term has been in use in the computer industry at least since the early 1960s. The form, design and implementation of CPUs have changed dramatically since the earliest examples, but their fundamental operation remains much the same.

The CPU, with the help of memory, executes instructions in the repetition of machine cycles. A *machine cycle* consists of four steps:

1. The control unit *fetches* an instruction and data associated with it from memory.
2. The control unit *decodes* the instruction.
3. The arithmetic/logic unit *executes* the instruction.
4. The arithmetic/logic unit stores the *result* in memory.

The first two instructions are called *instruction time, I-time*. Steps 3 and 4 are called *execution time, E-time*.

To a great extent a computer is defined by the power of its microprocessor. Chips with higher processing speed and more recent design offer the greatest performance and access to new technologies. Most microprocessors made for PCs are made by **Intel** or by companies that clone Intel chips, such as Advanced Micro Devices (**AMD**) and **Cyrix**.

The early Intel chip came in models called 286, 386, and 486. The 586 chip was given the name **Pentium**. The series of Pentiums were given the following names: **Pentium Pro**, **Pentium with MMX**, and **Pentium II**. The newer processors hold more transistors and thus more computing power on a single chip. It is made up of the control unit and the arithmetic and logic unit.

- **Arithmetic and Logic Unit (ALU)**

This unit consists of electronic circuits that perform the various arithmetic (calculations) and make the decisions (logic operations). The control unit is the master dispatcher of the computer system. It controls every activity in the computer system. It is indeed, the heart of the computer.

The speed of computer is measured in *Megahertz, MHz*. A MHz is a million machine cycles per second. A personal computer listed as 500 MHz has a processor capable of handling 500 million machine cycles per second. Another measure of speed is *Gigahertz (GHZ)*, a billion machine cycles per second. A third measure of speed is a *Megaflop*, which stands for one million floating-point operations per second. It measures the ability of the computer to perform complex mathematical operations.

- **Memory unit**

Memory, or primary storage, works with the CPU to hold instructions and data in order to be processed. Memory keeps the instructions and data for whatever programs you happen to be using at the moment. Memory is the first place data and instructions are placed after being input; processed information is placed in memory to be returned to an output device. It is very important to know that memory can hold data only temporarily because it requires a continuous flow of electrical current. If current is interrupted, data is lost. Memory is in the form of a semiconductor or silicon chip and is contained inside the computer.



There are two types of memory: **ROM and RAM**. **ROM** is read only memory. It contains programs and data that are permanently recorded when the computer is manufactured. It is read and used by the processor, but cannot be altered by the user. **RAM** is random access memory. The user can access data in RAM memory randomly. RAM can be erased or written over at will by the computer program or the computer user. The capacity of RAM has increased dramatically in recent years.

Memory is measured in **bytes**. A **byte** is usually made up of 8 bits and represents one character – a letter, digit, or symbol. The number of bytes that can be held is a measure of the memory and storage capacity. Bytes are usually measured in groups of kilobytes, megabytes, gigabytes, and terabytes. The following chart defines each term.

Kilobyte	KB	Roughly 1,000 bytes
Megabyte	MB	Roughly 1,000,000 bytes
Gigabyte	GB	Roughly 1,000,000,000 bytes
Terabyte	TB	Roughly 1,000,000,000 bytes

Memory is usually measured in Megabytes; a typical personal computer will have 64MB or more. Storage is usually measured in Gigabytes.

2.6.2 Storage Devices

A **storage device** is a hardware device designed to store information. There are two types of storage devices used in computers; a 'primary storage' device (the memory discussed above) and a 'secondary storage' device.

The secondary storage devices provide a permanent form of storage that does not depend on a constant flow of electricity. The benefits of secondary storage are large space capacity, reliability, convenience and economy.

The most common forms of Storage Devices found on your home computer are:

Floppy disk or Floppy	Hard disk (drive) or HD
<p>A round plastic surface that is coated with magnetic film. They come in 3^{1/2} size.</p> <p>They hold about 720k to 1440K of information. They are typically are used to install new software, save, share, and/or copy files. Floppy drives are given letters. Commonly the floppy is A, a 2nd floppy is B and the hard drive is C.</p>	<p>A hard disk is part of a unit, often called a "disk drive," "hard drive," or "hard disk drive," that store and provides relatively quick access to large amounts of data on an electromagnetically charged surface or set of surfaces.</p> <p>It is a stack of round metal platters called disks encased in a metal air tight shell. They commonly range in sizes from 1 to 250 gigabytes . The hard drive's function is to store all the files, and software the computer will ever use. Any file or software program used by RAM most likely will come from the disk drive.</p>
CD-ROM (Compact disk, read-only memory)	DVD-ROM (digital video disk, read-only memory)
<p>CD's function much like hard drive in that they store large amounts of memory. What separates them is their mobility and optical storage technology. Their storage capacity is also very limited compared to hard drives. The can only hold up to approximately 700 MB of information. The other big difference is that you have to have a special drive to write to CD's. Otherwise they can only be read from.</p>	<p>DVD's are similar to CD in that they are written and read by laser. Hard drives use magnetic currents store data. However CD's and DVD's use light (laser) to write and read data on a disk. These long and short pits are then stored or etched on the surface of the disk. They can only be read by laser technology. The new DVD technology increased the amount of memory a regular CD can hold. DVD's can range in sizes from 4.34GB (1000MB=1GB) to 7.95GB.</p>
USB Flash Drive	
<p>A flash drive is a storage module made of flash memory chips. A Flash disks have no mechanical platters or access arms, but the term "disk" is used because the data are accessed as if they were on a hard drive. The disk storage structure is emulated.</p> <p>It is a small, portable flash memory card that plugs into a computer's Universal Serial Bus (USB) port and functions as a portable hard drive. USB flash drives are touted as being easy-to-use as they are small enough to be carried in a pocket and can plug into any computer with a USB drive. USB flash drives have less storage capacity than an external hard drive, but they are smaller and more durable because they do not contain any internal moving parts. USB flash drives also are called</p>	

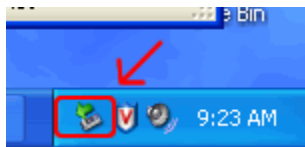
thumb drives, jump drives, pen drives, key drives, tokens, or simply USB drives. Flash drives come with varying amounts of storage capacity eg 128MB, 256, 4.5GB. Popular brand name drives, are Lexar, Sandisk, Kingston, PNY, Iomega and LG.



Other External storage devices include: Magnetic tape, Magnetic Disk, Tape Drive,

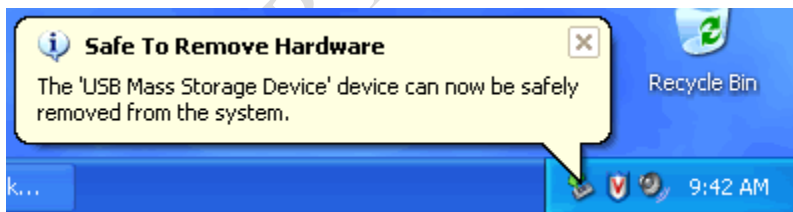
Working with Flash Drives

When the copying is finished, **do not immediately remove the flash drive from the USB port.**



Instead, left-click on the **Remove Hardware** icon located in the System Tray. A window containing a list of the USB devices will appear. Left-click on the **Safely Remove Mass Storage Device** line that matches your flash drive (for example, **Safely Remove Mass Storage Device - Drive(G:)**).

When you see the following message appear in the bottom left toolbar, it is, as it says, safe to remove the flash drive from the USB port; you may close the message or ignore it, as it will close itself automatically:



Summary

You have been introduced to computers in this session. The general characteristics and usefulness of computers, the basic architecture and the basic hardware components of a component were discussed. Now answer the following questions.

Self Assessment Questions

1. What is a computer? Give five characteristics of a computer.
2. Describe the basic architecture of a computer
3. Compare and contrast any three storage devices

PROPERTIES OF DLC UI, IBADAN

Study Session 3: Other Hardware Components of Computers

Expected Duration: 1 week or 2 contact hours

Introduction

In this Session, you will learn about the other hardware components of computers. The different hardware components like the motherboard, cards and expansion slots are explained in this session. The session ends with the knowledge of booting computers for use.

Learning Outcomes

When you have studied this session, you should be able to explain:

- 3.1 The Motherboard
- 3.2 Expansion slots
- 3.4 Sound card
- 3.5 Network Interface Cards (NICs)
- 3.6 PC Card
- 3.7 ExpressCard
- 3.8 Ports
- 3.9 USB
- 3.10 Firewire
- 3.11 Thunderbolt
- 3.12 Ethernet
- 3.13 Computer Booting Activities

3.1 The Motherboard

The components inside a computer need to be able to talk to each other. A motherboard is an electronic circuit board in a computer which interconnects the hardware that is attached to it. At a minimum, it connects a CPU and memory. Sometimes there are smaller processors that help take some of the load off of the CPU; busses process destinations for data, so the CPU can be left to do what it does best.

A motherboard normally has a set of expansion slots, which allow the motherboard to be expanded; it can be given extra functionality that it didn't have originally. Smaller boards called *cards* fit in these expansion slots, and these cards contain specialized circuits that let the motherboard do more. Typical motherboards also have a series of sockets, I/O, allowing communication through cables with various peripheral devices, both inside and outside the computer case.

3.2 Expansion slots

These are connectors on a motherboard where a circuit board can be inserted to add new capabilities. All personal computers contain expansion slots for adding more RAM, graphics capabilities, and support for special devices; some are more specialized than others.

3.3 Graphics card



Graphics cards are also called video cards or sometimes video adapters. They are in most PCs. Graphic cards convert data signals into video signals so the images can be displayed on the monitor. While graphics processors are often built into a motherboard and a card isn't needed, graphics cards have stronger and more powerful processing capabilities that allow the display of 3d images, heavy image editing, or better rendering and frame rates in computer games.

Graphics cards are designed to offload the burden of making images from the CPU. Graphics cards also include on board memory for efficient rendering. Typical sizes include 128-1024mb of memory. Today, high end graphics cards have multiple core processors that are largely parallel to increase texture fill and process more 3D objects in real time.

3.4 Sound card



A sound card, also referred to as an audio card, allows the input and output of audio signals to and from a computer under the control of computer programs. Sound cards for computers were unusual to find until 1988, until then, the single internal PC speaker was the only way early PC software could produce sound and music. Uses of a sound card include the audio components for multimedia applications such as games, video/audio editing software and music composition. Most computers today have sound capabilities built into the motherboard, while others require additional expansion cards.

3.5 Network Interface Cards (NICs)

A network interface card can be a network card, network adapter, LAN Adapter or NIC (network interface card). They are a piece of computer hardware designed to allow computers to communicate over a computer network.

3.6 PC Card

A PC Card (originally PCMCIA Card) is any card, like a graphics or network card, designed for laptop computers. Laptops have a different style of expansion slot called a PCMCIA slot. PC cards are being phased out, and are usually only found on older laptops.

3.7 ExpressCard

ExpressCards are replacing PC cards in modern computers because of their smaller size.

3.8 Ports

Ports are used by a motherboard to interface with electronics; a way of getting input and output to and from hardware. Ports are commonly used to attach peripherals, specifically, hardware that exists outside the computer's case, although some ports connect to things inside the computer too. The amount of data that can be transferred over a port during a given amount of time is its bandwidth. Parallel ports are used to connect external hardware such as printers, scanners, or fax machines. While this technology is slowly being phased out in favour of USB, Parallel ports can still be found in many motherboards today.

3.9 USB

USB is an acronym for universal serial bus. USB was designed to standardize the connection of computer peripherals, such as keyboards, pointing devices, digital cameras, printers, portable media players, disk drives and network adapters to personal computers, both to communicate and to supply electric power. It has become commonplace on other devices, such as smartphones, PDAs and video game consoles. USB has effectively replaced a variety of earlier interfaces, such as serial and parallel ports, as well as separate power chargers for portable devices.

3.10 Firewire

Technically known as the IEEE 1394 interface, but dubbed by Apple as Firewire, this connection medium hoped to surpass USB in terms of speed and popularity. While it did outperform USB v2 in speed tests, uptake was very limited due to the existing widespread use of USB.

3.11 Thunderbolt

A recently developed port called Thunderbolt was released by Intel. It's expected to be a strong competitor to USB, because it can transfer a large amount of information at one time.

3.12 Ethernet

Ethernet ports are typically found on NICs. It is a network port allowing data transfer from one computer to another. Some printers have network capabilities with ethernet ports.

3.13 Computer Booting Activities

Booting is the process of loading an operating system into computer's memory. A computer cannot load more than one operating system at a time when turned on.

- When a computer is turned on, a program called diagnostic routine tests the main memory, the CPU, and other system parts to make sure they are working properly.
- Computer then loads small set of instructions, ROM (Read Only Memory), from BIOS chips which is a non-volatile memory.
- BIOS programs are copied into the main memory. It helps the computer to interpret or transmit characters to the monitor, diskette or Compact Disk.
- The operating system is obtained from the hard disk and loads into the main memory. The operating system remains there until you turn off your system.

- User interface is activated. It is the first thing you see after booting. The user-interface allows the users to communicate, or interact with the computer.
- Then the system is ready to use.

3.13.1 Types of Booting:

(i) Cold boot: This is when you power-on the computer through the switch button.

(ii) Warm Boot or Warm Start: This is when you restart the computer. This can be done by pressing Ctrl+Alt+Del keys or pressing restart button or choosing restart option from the start menu.

Summary

In this Session, the following hardware sub-components were discussed.

- The Motherboard
- Expansion slots
- Sound card
- Network Interface Cards (NICs)
- PC Card
- ExpressCard
- Ports
- USB
- Firewire
- Thunderbolt
- Ethernet
- Computer Booting Activities

Self-Assessment Questions

1. Explain the usefulness of the following computer hardware components

1. The Motherboard
2. Network Interface Cards (NICs)
3. Ethernet

2. Explain the meaning of cold and warm boots in computer start-ups

Study Session 4: Software Components of Computers

Expected Duration: 1 week or 2 contact hours

Introduction

Software is a generic term for organized collections of computer data and instructions, often broken into two major categories: system software that provides the basic non-task-specific functions of the computer, and application software which is used by users to accomplish specific tasks. In this Session, you shall be introduced to the software components of computers.

Learning Outcomes

When you have studied this session, you should be able to explain:

- 4.1 The Software Subcomponent
 - 4.1.1 Application Software
 - 4.1.2 System Software
 - 4.1.3 Programming Software
- 4.2 Operating Systems
- 4.3 Operating System Types

4.1 The Software Subcomponent

Software is a set of electronic instructions that tell a computer what to do. As important as hardware devices may be, they are useless without the instructions that control them. These instructions used to control hardware and accomplish tasks are called *software*. Software generally refers to all the programs that you can run on the computer hardware. Software falls into three broad categories— Applications, System and Program Software.

4.1.1 Application Software

Application software is used to accomplish specific tasks other than just running the computer system. Application software may consist of a single program, such as an image viewer; a small collection of programs (often called a software package) that work closely together to accomplish a task, such as a spreadsheet or text processing system; a larger collection (often called a software suite) of related but independent programs and packages that have a common user interface or shared data format, such as Microsoft Office, which consists of closely integrated word processor, spreadsheet, database, etc.; or a software system, such as a database management system, which is a collection of fundamental programs that may provide some service to a variety of other independent applications.

You can use application software to write letters, manage your finances, draw pictures, play games and much more. Application software is also called packages, an application or a program. Packaged software can be purchased, or in some cases, it is available for no cost. *Freeware* is software considered to be in the public domain, and it may be used or altered without fee or restriction. Another form of somewhat free software is *shareware*. The author of shareware hopes you will make a voluntary contribution for using the product.

Software Category	Function
Word Processor	Provides the tools for entering and revising text, adding graphical elements, formatting and printing documents. E.g. MS Word
Spreadsheets	Provides the tools for working with numbers and allows you to create and edit electronic spreadsheets in managing and analyzing information. Eg MS Excel
Database Management	Provides the tools for management of a collection of interrelated facts. Data can be stored, updated, manipulated, retrieved, and reported in a variety of ways. e.g MS Access
Presentation Graphics	Provides the tools for creating graphics that represent data in a visual, easily understood format. Eg MS PowerPoint
Communication Software	Provides the tools for connecting one computer with another to enable sending and receiving information and sharing files and resources.
Internet Browser	Provides access to the Internet through a service provider by using a graphical interface. Internet Explorer
Accounting Packages	These are application software that cover all accounting functions such as sales ledger, payroll, trial balance sheet, costing etc. Eg. Peachtree, Dac Easy
Statistical Packages	<i>Statistical Packages:</i> These are packages that can be used to solve statistical problems e.g SPSS (Statistical Package for Social Science), SAS (Statistical Analysis System), Stata, Eviews, etc
Desktop Management	Packages that facilitate publishing. e.g Corel draw, PageMaker

Graphic Packages	These are designed for producing images, diagrams, and various picture e.g, Havard Graphics, Graph Writer.
Games/Entertainment	These packages allow the computer user to (children or adult) play different variety of games on a computer e.g Star trek, Chess, Football, War games etc.
Audit Packages	They are design for external auditors, and internal auditors e.g. Audit command language
Tutorials	These are group of Packages tailored toward teaching on software or hardware

4.1.2 System Software

System software is responsible for controlling, integrating, and managing the individual hardware components of a computer system so that other software and the users of the system see it as a functional unit without having to be concerned with the low-level details such as transferring data from memory to disk, or rendering text onto a display.

The “System Software” are fundamental programs that run in any computer and are usually written by system programmers and in addition, are hardware related. The systems Software manage computer resources and interact with both the user and application programs. System software helps run the computer hardware and computer system. It includes:

- device drivers,
- diagnostic tools,
- operating systems,
- servers,
- utilities,
- windowing systems,

The purpose of systems software is to insulate the applications programmer as much as possible from the details of the particular computer complex being used, especially memory and other hardware features, and such as accessory devices as communications, printers, readers, displays, keyboards, etc.

4.1.3 Programming Software

Programming software usually provides tools to assist a programmer in writing computer programs, and software using different programming languages in a more convenient way. The tools include:

- compilers,
- debuggers,

- interpreters,
- linkers,
- text editors,

4.2 Operating Systems

An operating system is a software component of a computer system that is responsible for the management of various activities of the computer and the sharing of computer resources. It hosts the several applications that run on a computer and handles the operations of computer hardware. Users and application programs access the services offered by the operating systems, by means of system calls and application programming interfaces. Users interact with operating systems through Command Line Interfaces (CLIs) or Graphical User Interfaces known as GUIs. Generally, operating system enables user interaction with computer systems by acting as an interface between users or application programs and the computer hardware.

4.3 Operating System Types

There are many types of operating systems. The most common is the Microsoft suite of operating systems. They include from most recent to the oldest:

- Windows Vista- The latest version in the Microsoft suite of operating systems with more new features for searching, staying connected, networking, syncing devices, security and managing files on your computer.
- Windows XP Professional Edition - A version used by many businesses on workstations. It has the ability to become a member of a corporate domain.
- Windows XP Home Edition - A lower cost version of Windows XP which is for home use only and should not be used at a business.
- Windows 2000 - A better version of the Windows NT operating system which works well both at home and as a workstation at a business. It includes technologies which allow hardware to be automatically detected and other enhancements over Windows NT.
- Windows ME - A upgraded version from windows 98 but it has been historically plagued with programming errors which may be frustrating for home users.
- Windows 98 - This was produced in two main versions. The first Windows 98 version was plagued with programming errors but the Windows 98 Second Edition which came out later was much better with many errors resolved.
- Windows NT - A version of Windows made specifically for businesses offering better control over workstation capabilities to help network administrators.
- Windows 95 - The first version of Windows after the older Windows 3.x versions offering a better interface and better library functions for programs.

Find out about latest Windows around now.

There are other worthwhile types of operating systems not made by Microsoft. The greatest problem with these operating systems lies in the fact that not as many application programs are written for them. However if you can get the type of application programs you are looking for, one of the systems listed below may be a good choice.

- (a) **Unix** - A system that has been around for many years and it is very stable. It is primary used to be a server rather than a workstation and should not be used by anyone who does not understand the system. It can be difficult to learn. Unix must normally run on a computer made by the same company that produces the software.
- (b) **Linux** - Linux is similar to Unix in operation but it is free. It also should not be used by anyone who does not understand the system and can be difficult to learn.

Summary

You have been introduced to the software component of a component. Application and System software are the major types of software. Can you mention the different types of each of these software components?

Self-Assessment Questions

1. Differentiate with copious examples, the difference between application and system software.
2. Discuss the following slogan: “No operating system, no computer”.

Study Session 5: Generations of Hardware and Software

Expected Duration: 1 week or 2 contact hours

Introduction

In this session, we discussed the generation and / or evolution of computer hardware and software. The major characteristics of each generation were pointed out.

Learning Outcomes

When you have studied this session, you should be able to explain:

- 5.1 History of Hardware Generation
- 5.2 Trends of Computer Technology
- 5.3 Three Directions of Computer Development
- 5.4 Interpreting Computer Specifications
- 5.5 Computer Booting Activities
- 5.6 History of Software Generation

5.1 History of Hardware Generation

(i) First Generation (1940-1956): Vacuum Tubes

The first computers used vacuum tubes for circuitry and magnetic drums for memory and were often enormous, taking up entire rooms. They were very expensive to operate and in addition to using a great deal of electricity, generated a lot of heat, which was often the cause of malfunctions, input was based on punched cards and paper tape, and output was displayed on printouts. The UNIVAC and ENIAC computers are examples of first-generation computing devices. The ENIAC was one of the first computers delivered to the U.S. Army in 1946, while UNIVAC was the first commercial computer delivered to a business client, the U.S Census Bureau 1951.

(ii) Second Generation (1956-1963) Transistors

Transistors replaced vacuum tubes and ushered in the second generation of computers. The transistor was invented in 1947, but did not see widespread use in computers until the late 1950s. The transistors was far become smaller, faster, cheaper more energy-efficient and more reliable than their first-generation predecessors. Though the transistor still generated a great deal of heat that subjected the computer to damage, it was a vast improvement over the vacuum tube. Second-generation computers still relied on punched cards for input and printouts for output. These were also the first computers that stored their instructions in their primary memory, which moved from a magnetic drum to magnetic core technology. The first computers of this generation were developed for the atomic energy industry.

(iii) Third Generation (1964-1971) Integrated Circuits

The development of the integrated circuit was the hallmark of the third generation of computers. Transistors were miniaturized and placed on silicon chips, called semi-conductors, which drastically increased the speed and efficiency of computers. Instead of

punched cards and print-outs, users interacted with third generation computers through keyboards and monitors and interfaced with an operating system, which allowed the device to run many different applications at one time with a central program that monitored the memory. Computers for the first time became accessible to a mass audience because they were smaller and cheaper than their predecessors.

(iv) Fourth Generation (1971-Present) Microprocessors

The microprocessors brought the fourth generation of computers, as thousands of integrated circuits were built onto a single silicon chip. What filled an entire room in the first generation could now fit in the palm of a hand. The Intel 4004 chip, developed in 1971, located all the components of a central Processing unit on a single chip. In 1981 IBM introduced its first computer for the home user, and in 1984, Apple introduced the Macintosh. Micro-processors also moved out of the realm of desktop computers and into many areas of life as more and more everyday products began to use micro-processors. As these small computers became more powerful, they could be linked together to form networks, which eventually led to the development of the Internet. Fourth generation computers also saw the development of GUIs (Graphical User Interface), the mouse and handled devices.

(v) Fifth Generation (Present and Beyond) Artificial Intelligence

In 1981, Japan announced its focus on the fifth generation of computers to be realized by 1990. The machine is supposed to have the following characteristics:

- (a) Integrated circuits with a minimum of 1 million transistors per semiconductor chip.
- (b) The ability to communicate with users in natural language; spoken, written and graphic languages will be accepted with the ability to translate among the three.
- (c) The ability to accumulate knowledge so that the system appears to “learn and infer”. This enables the system to answer vague and unanticipated requests and to make decisions to augment human decisions.
- (d) Simplified programming with the use of more structured designed that will help users define their own problems and be able to generate reliable programs to solve them.
- (e) A variety of sizes from portable to supercomputer that can operate within a network structure

Fifth generation computing devices, based on artificial intelligence, are still in development, though there are some applications, such as voice recognition, that are being used today. The use of parallel processing and superconductors is helping to make artificial intelligence a reality. Quantum computation and molecular and nanotechnology will radically change the face of computers in year to come. The goal of fifth-generation computing is to develop devices that respond to natural language input and are capable of learning and self-organization. The main objective is to rely less heavily on existing software, which currently depends on professional skills and expertise.

5.2 Trends of Computer Technology

- 1621CE -1642 - Slide rule invented (Edmund Gunther)
- 1642-1833 - First mechanical adding machine (Blaise Pascal)
- 1833-1843 - Babbage's difference engine (automatic calculator)
- 1843-1890 - World's first computer programmer, Ada Lovelace, published her first Notes.
- 1890-1930 - Electricity used for the first time in a data processing project (punched cards)
- 1930-1944 - General theory of Computer
- 1944-1946 - First electromechanical computer (MARK 1)
- 1946-1952 - First programmable electronic computer in the United States (ENIAC)
- 1952-1964 - UNIVAC computer correctly predicts election of Eisenhower as U.S President
- 1964-1967 - IBM introduces 360 lines of computers
- 1967-1969 - Handheld calculator
- 1969-1970 - ARPANet established and this led to the Internet.
- 1970-1971 - Microprocessor chips come into use and floppy disk was introduced for storing data.
- 1971-1973 - First pocket Calculator.
- 1973-1975 - FTP (File transfer protocol) was developed
- 1975-1976 - First micro-computer
- 1976-1978 - Apple 1 computer (first personal computer sold in assembled form) with 512B RAM
- 1978-1981 - 5¼ floppy disk
- 1981-1982 - IBM introduces personal computer
- 1982-1984 - Portable computers; TCP/IP was established as an Internet standard. The Word Internet was coined.
- 1984-1993 - Apple Macintosh, first personal laser printer, desktop publishing started, Domain Name (DNS) is introduced.
- 1993-1994 - Multimedia desktop computers, i.e. personal digital assistance
- 1994-1997 - Apple and IBM introduces PCs with full-motion video built in, inbuilt wireless data transmission for small portable computers, and web browser, Mosaic was invented.
- 1994-1997 - Network Computers and Pathfinder robot lands on the Mars.
- 1998-2000 - Digital HDTV broadcasts begin
- 2000-2001 - Microsoft NET announced
- 2001-2003 - Windows XP and Mac OS X
- 2003-till date - Mac G5 and experts predict that by 2047 all electronically encodable information will be in cyberspace.

5.3 Three Directions of Computer Development

ENIAC computer was the first programmable general-purpose computer in the USA. ENIAC is the grandparent of today's lightweight computers. ENIAC's weight was 30 tons and was 80 feet long and two stories high. Since the days of ENIAC, computers have been developing in three major directions which are:

(i) **Miniaturization:**

Computer hardware technology improves from vacuum tubes to transistors, from transistors to integrated circuits (ICs). ICs are solid pieces of silicon that contained the transistors, other components and their connections. ICs have scaled from large scale integrated circuits to Very Large Scale Integrated circuits (VLSI). This makes the size of the computer to get smaller. Thus a laptop computer of a miniaturized processor can perform calculations that once required a computer whose size filled an entire room.

(ii) **Speed:**

Miniaturization of the processor and new materials for making the processor have made it possible to cram more hardware devices into smaller computer machines, resulting in faster processing speed and increased storage capacity.

(iii) **Affordability:**

The use of silicon which is a very cheap material got from sand makes the price of the processor to be cheap. Thus the price of the computer machine is becoming cheaper as technology improves. Therefore, individuals and businesses can own their own computers.

5.4 Interpreting Computer Specifications

Given the following computer hardware specification:

1. Pentium R-Dual core @ 2.20GHz @2.20 GHz
2. 3G DDR-SDRAM at 400MHz
3. 300G Ultra ATA/100 HARD Drive
4. 17" Flat-Panel Display
5. 8X DVD+R/RW Drive with CD-RW
6. Altec Lansing Surround Sound Speakers
7. Integrated 5.1 Audio with Dolby Digital
8. 56K PCI Data/Fax Modem
9. WordPerfect® productivity pack

Interpretations of the above

- Pentium R-Dual core @2.20GHz @2.20GHz indicates the type of processor. Dual means two and the rate of transfer, i.e., speed of access of each processor is 2.20GHz.
- 3G means 3 gigabytes, which is the capacity of the RAM. This RAM can contain up to 3G worth of information that can be directly accessed.
- DDR-SDRAM indicates the type of RAM. DDR-SDRAM means double-data rate synchronous dynamic RAM. It is the newest RAM and mostly common RAM in notebook computers.
- 400MHz indicates speed, i.e., the memory can be accessed at 400×10^6 per second.
- 300G Ultra ATA/100 HARD Drive: 300G indicates the size of the hard disk, i.e., the secondary storage facility. Ultra ATA (Advance Technology Attachment) indicates the type of hard disk controller. Another type of controller SCSI (Small Computer

System Interface). Disk controller positions the hard disk and read/write heads and manages flow of data and instructions to and from the disk. Ultra ATA supports only two hard disks while SCSI supports several disks and other peripherals up to seven.

- 17" Flat-Panel Display indicates the diagonal measurement of the monitor screen. In this case, the visible part can be 16"

The choice of computer specification depends on what you want to use it for. For example, if you need a computer for graphic work, then you need one with good memory because graphic work and software are very heavy.

5.5 Computer Booting Activities

Booting is the process of loading an operating system (OS) such as Windows or Linux into computer's memory. A computer cannot load more than one operating system at a time when turned on.

- (i) When a computer is turned on, a program called diagnostic routine tests the main memory, the CPU and other system parts to make sure they are working properly.
- (ii) Computer then loads small set of instructions, ROM (Read Only Memory) from BIOS chip, which is a non-volatile memory.
- (iii) BIOS programs are copied into the main memory. It helps the computer to interpret or transmit characters to the monitor, diskette or Compact Disk.
- (iv) The OS is obtained from the hard disk and loads into the main memory. The OS remains there until the computer is turned off.
- (v) User interface is activated. It is the first interface seen after booting. The user interface allows users to communicate or interact with the computer.
- (vi) The system is now ready for use.

5.5.1 Types of Booting

We have two types of booting:

- (i) **Cold Booting.** This is when the computer is powered –on via the switch button.
- (ii) **Warm Booting or Warm Start.** This occurs when the computer is re-started by pressing CTRL + ALT + DEL keys or pressing restart button or choosing restart option from the start menu

Class Discussion

- A student using Microsoft Word to type class assignment without saving has the unsaved work on the RAM; true or false? If your answer is true, what does this imply? If your answer is false, what does this imply? If your answer is false, explain.
- Discuss at least two of the applications of fifth generation computers.

5.6 History of Software Generation

The Hardware of a computer cannot drive itself even though it is turned on. It needs a program that will direct it on what to do and how to do it. Thus we need software. The historical development of Computer Software is discussed below.

- (i) **First generation (1951-59).** This generation uses machine language written in binary format, i.e. (1's and 0's). Programmers had to be very good with numbers and very detailed oriented. First generation programmers were mathematicians and engineers. Each CPU has its own machine language; therefore, it varies from computer to computer. Machine code was tedious and error prone.
- (ii) **Second generation (1959-65):** To reduce the problems of First Generation Software, assembly language that uses mnemonics was developed. It is a low-level language which uses abbreviation or more easily remembered numbers. It is also machine dependent. Therefore, there is a need for a translator called the **Assembler**. Assembler is a program that reads program instructions written in mnemonic form and translates into machine language.
- (iii) **Third generation (1965-71):** As hardware became more powerful, more powerful software tools were needed. High level or Procedural languages that are English – like were developed. Examples are FORTRAN (the first High-level language), COBOL, and Basic. Most of these languages are machine independent. These languages also need translators. The translators are compiler and interpreter. For example FORTRAN and COBOL use compiler, while BASIC uses interpreter. Software Operating Systems (OS) was developed. OS controls/manages Computer resources.
- (iv) **Fourth Generation (1971-1989):** Very high-level or problem-oriented languages were developed. These languages allow users to develop programs with fewer commands and better structures. They have report generations, query languages (an easy-to-use language) for retrieving data from database management system, e.g. SQL and Intellect, and application generators (this was used to develop application packages). More advancement was made on third generation languages for structured programming, e.g. Refined BASIC language, C, C++, etc. Also Object-Oriented Programming (OOP) was developed. Better and more powerful OS was developed.
- (v) **Fifth generation (1990-present):** Declarative programming languages were developed. These are programming languages that use problem descriptions to give people a more natural connection with computers. They are part of the field of study known as artificial intelligence. There was the rise of Microsoft as a major player in software industry, Object Oriented Design and Programming (Java) were developed and World Wide Web (WWW) was introduced on the Internet. Application packages (Spreadsheet, Word Processing, Database programs, etc.) were bundled together into super packages called Office suite.

In summary, first and second generations' languages were more technical and less user friendly, while third, fourth and fifth generations become less technical and more user friendly as we approach the future.

Summary

Self-Assessment Study Questions

1. In a tabular form compare and contrast the generation hardware with respect to size, speed, primary memory and affordability.
2. Name the components of a von Neumann machine
3. Compare and contrast RAM and ROM
4. Distinguish between cold boot and warm boot.
5. When the computer is not on, the operating system resides in the
6. What is a secondary storage device, and why are such devices important?
7. What are the two basic secondary storage media? Distinguish between the two and give examples.
8. Differentiate the following
 - CD-ROM
 - CD-R
 - CD-RW
 - DVD-ROM
 - DVD-R or DVD+R
 - DVD-RW or DVD+RW
9. What are the three directions of computer development? Explain.
10. Explain the most important landmarks in the history of software development

Study Session 6: Introduction to Computer Networks and Internet

Expected Duration: 1 week or 2 contact hours

Introduction

Connecting two or more computers together via some equipment is the order of the day. We use internet and other networks for transmitting and receiving information today. This is made possible because of the availability of computer networks. In this Session, you shall be introduced to the concept of computer networks, the different equipment needed and styles of connecting the network, called topology.

Learning Outcomes

When you have studied this session, you should be able to explain:

- 6.1 The meaning of a network
- 6.2 The meaning of networking
- 6.3 The Meaning of internetworking
- 6.4 The internetworking devices
- 6.5 Network Topology
- 6.6 The Internet

6.1 What is a network?

Network (in Information Technology) is a series of points or nodes interconnected by communication path for the purpose of computer resource and file sharing

6.2 What is networking?

Networking is the construction, design and use of network, including the physical (cabling, hub, bridge, switch, routers, and so forth), the selection and use of telecommunication protocol and computer software for using and managing the network, and establishment of operating policies and procedures related to network.

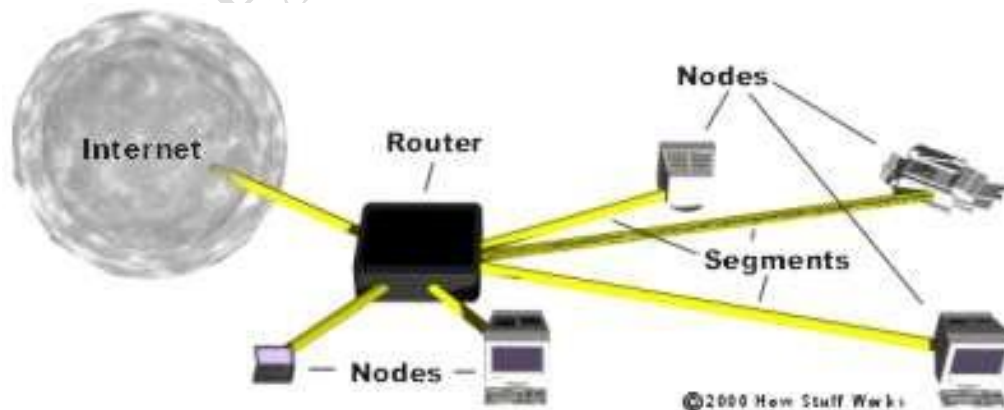


Figure 6.1 Networking basic elements (Source: How stuff works)

6.3 What is an internetworking?

Internetworking is a collection of individual networks, connected by intermediate devices (e.g. routers), that function as a single large network. Internetworking refers to the industry, product and procedure that meet the challenge of creating and administering internetworks. The **figure below** illustrates some different kinds of network technologies that can be interconnected by routers and other networking Devices to create an internetwork.

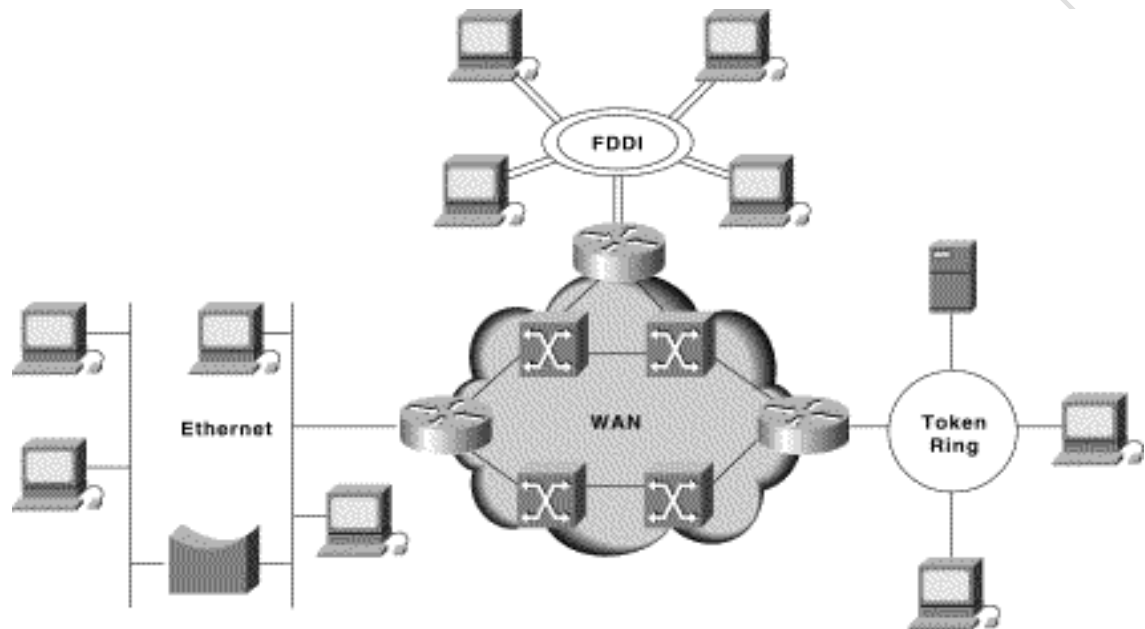


Figure 6.2: Different network technologies connected to create an internetwork.

6.4 Internetworking devices

These are devices used in connecting individual networks to each other. Examples include:

(i) Cables

Cables play important roles in networking and the type of network media used determines how fast the data travels along the media, and also the maximum data rate that can be carried. The four main types of networking media are:

(ii) Shielded twisted-pair:

This is a type of copper telephone wiring in which each of the two copper wires that are twisted together and are coated with an insulating coating that functions as a ground for the wires.

(iii) Unshielded twisted-pair:

Unshielded Twisted-pair are not shielded and thus interfere with nearby cables. They are used in LANs to bit rates of 100Mbps and with maximum length of 100m. UTP cables are typically used to connect a computer to a network.

(iv) Coaxial cable:

Coaxial cable has a grounded metal sheath around the signal conductor. Interference among cables is reduced due to the sheath around signal conductor. The cable allows higher data rate transfer. Typically they are used at bit rates of 100 Mbps for maximum lengths of 1 km.

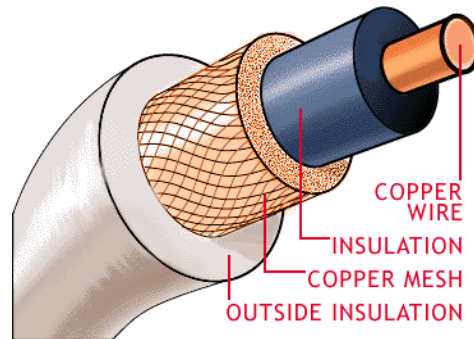


Figure 6.3: Coaxial cable

(v) Fiber-optic cable:

These are made of tiny fiber of glass, capable of reflecting signals. Fiber optic cables do not interfere with nearby cables. They give greater security. They allow extremely high bit rates over long distances. They provide more protection from electrical damage by external equipment and greater resistance to harsh environment. They are safer in hazardous environment



Figure 6.4: Fiber Optic cable

(vi) Wireless LANs



Figure 6.5: Wireless LAN connection

Not all networks are connected with cabling; some networks are wireless. Wireless LANs use high frequency radio signals, infrared light beams, or lasers to communicate between the workstations and the file server or hubs. Each workstation and file server on a wireless network has some sort of transceiver/antenna to send and receive the data. Information is relayed between transceivers as if they were physically connected. For longer distance, wireless communications can also take place through cellular telephone technology, microwave transmission, or by satellite.

Wireless networks are great for allowing laptop computers or remote computers to connect to the LAN. Wireless networks are also beneficial in older buildings where it may be difficult or impossible to install cables.

The two most common types of infrared communications used in schools are line-of-sight and scattered broadcast. Line-of-sight communication means that there must be an unblocked direct line between the workstation and the transceiver. If a person walks within the line-of-sight while there is a transmission, the information would need to be sent again. This kind of obstruction can slow down the wireless network.

Scattered infrared communication is a broadcast of infrared transmissions sent out in multiple directions that bounces off walls and ceilings until it eventually hits the receiver. Networking communications with laser are virtually the same as line-of-sight infrared networks.

Wireless LANs have several disadvantages. They provide poor security, and are susceptible to interference from lights and electronic devices. They are also slower than LANs using cabling.

6.4.1 Installing Cable - Some Guidelines

When running cable, it is best to follow a few simple rules:

- (a) Always use more cable than you need. Leave plenty of slack.
- (b) Test every part of a network as you install it. Even if it is brand new, it may have problems that will be difficult to isolate later.

- (c) Stay at least 3 feet away from fluorescent light boxes and other sources of electrical interference.
- (d) If it is necessary to run cable across the floor, cover the cable with cable protectors.
- (e) Label both ends of each cable.
- (f) Use cable ties (not tape) to keep cables in the same location together.

(vii) Repeaters

As a network grows, its cable can easily exhaust the network constraint, and any further loads on this network can lead to attenuation and digital pulse distortion. To overcome these limitations, repeaters are used to increase the maximum interconnection length, and may do the following:

- (a) Reshape signal pulses
- (b) Pass all signals between attached segments
- (c) Boost signal power
- (d) Possibly translate between two different media (such as between fiber-optic and twisted-pair cable)
- (e) Transmit to more than one network. These are multi-port repeaters and send data frames from any received segment to all the others. Multi-port repeaters do not filter the traffic, as they blindly send received data frames to all the physical connected network segments.

(viii) Hubs

A hub is a repeater with multiple ports, and can be thought of as being the centre point of a star topology network. It is often known as a multi-port repeater (or as a concentrator in Ethernet). Hubs can be active (where they repeat signal sent through them) or passive (where they do not repeat, but merely split, signals sent through them). Hub generally:

- Amplify signals.
- Propagate the signal through the network.
- Do not filter traffic. This is a major disadvantage with hubs and repeaters as data arriving at any of the ports is automatically transmitted to all the other ports connected to the hub.
- Do not determine path.
- Centralize the connection to the network

Figure 1a illustrates a star topology with hub as the central server and fig. 1b shows the real hub as used in networking.

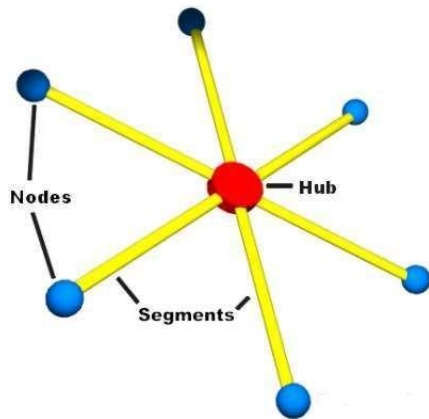


Figure 6.6a: A pictorial hub



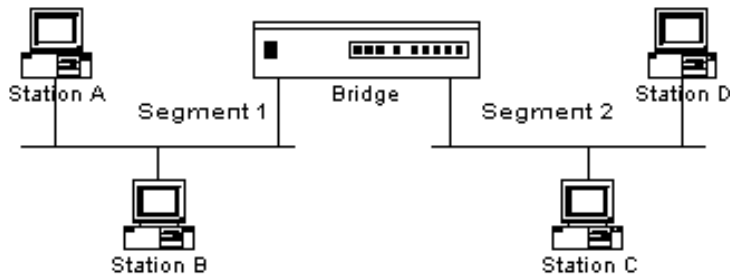
Figure 6.6b: A typical hub

(ix) Bridges

These examine the destination MAC address (or station address) of the transmitted data frames, and will not retransmit data frames which are not destined for another network segment.

They maintain a table with connected MAC address, and do not forward any data frames if the MAC address is on the network that originated it, else it forward to all connected segment. Thus a bridge does not actually determine on which segment the destination is on, and blindly forwards data frames to all other connected segments.

Bridges are an excellent method of reducing traffic on network segment and it does this by examining the destination address of the frame before deciding how to handle it. Consider the figure below; if the destination address of a data frame (from segment 1) is that of station A or B, then there is no need for the frame to appear on segment 2. But if the destination address (data frame from segment 1) is that of station C or D, or if it is the broadcast address, then the bridge will transmit, or forward the frame on to segment 2. By forwarding packets, the bridge allows any of the four devices in the figure to communicate.



©2001 HowStuffWorks

Figure 6.7: An Ethernet Bridge Connecting Two Segments

Problems with Bridge

- (a) They work well when there is not too much inter-segment traffic, but when the inter-segment traffic becomes too heavy the bridges can actually become a bottleneck for traffic, and actually slow down communication.
- (b) They spread and multiply broadcast. A bridge forwards all broadcast to all other connected segments. If there are too many broadcast, it can result in a broadcast storm.

(x) Switches

A switch is a very fast, low-latency, multi-port bridge that is used to segment LANs. They are typically used to increase communication rates between segments with multiple parallel conversation and also communication between different networking technologies (such as between ATM and 100Base-TX).

A vital difference between a hub and a switch is that all the nodes connected to a hub share the bandwidth among themselves, while a device connected to a switch port has the full bandwidth all to itself. For example, if 10 nodes are communicating using a hub on a 10-Mbps network, then each node may only get a portion of the 10 Mbps if other nodes on the hub want to communicate as well. But with a switch, each node could possibly communicate at the full 10 Mbps.

Figure 6.8 (a) represents a typical switch and (b) illustrates a network using a switch.



Figure 6.8 (a) A Switch

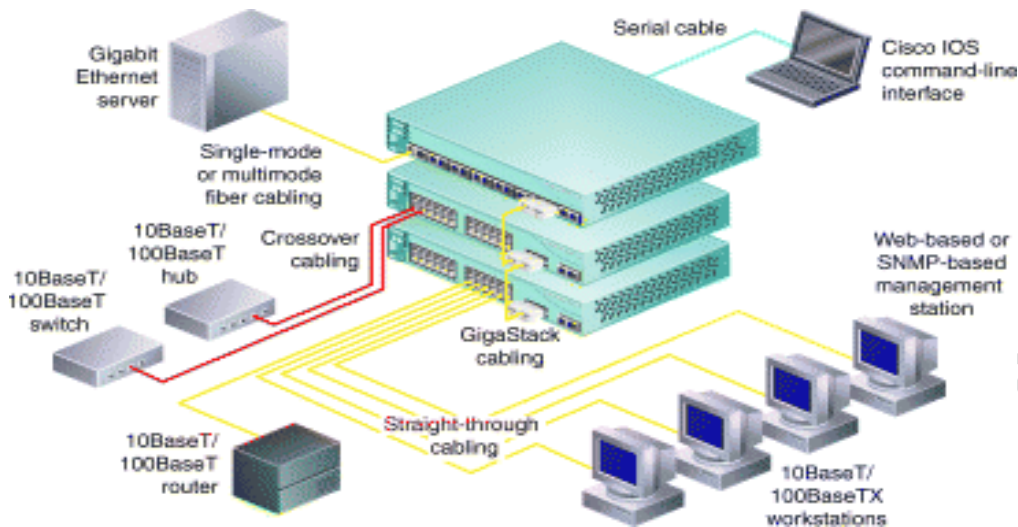


Figure 6.8 (b) A network using a switch

Switches forwards received data frames in two ways;

- (a) **Cutting-through switching:** the switch reads the destination address before receiving the entire frame. The data is then forwarded before the entire frame arrives. This method has the advantage that there is less delay (latency) between the reception and transmission of a data packet, but has poor error detection, because it does not have a chance to detect any errors, before it has started to transmit the received data frame.
- (b) **Store-and-forward switching:** it involves reading the entire Ethernet frame, before forwarding it, with the required protocol and at the correct speed, to the destination port. This has the advantage of improved error check, protocol filtering and speed matching but has the disadvantage of system delay, as the frames must be totally read before it is transmitted. r

(xi) Routers

These are network devices that examine the network address (IP address) field and determine the best route for a data packet, and will only transmit it out of a network segment if it is destined for a node on another network. Routers do the following:

- (a) Do not forward broadcast
- (b) Do not forward traffic to unknown addresses
- (c) Modify data packet header
- (d) Build tables of network addresses.

6.5 Network Topology

A network consists of multiple computers connected using some type of interface, each having one or more interface devices such as a Network Interface Card (NIC) and/or a serial device for PPP networking. Each computer is supported by network software that provides the server or client functionality. The hardware used to transmit data across the network is called the media. It

may include copper cable, fiber optic, or wireless transmission. The standard cabling used for the purposes of this document is 10Base-T category 5 ethernet cable. This is twisted copper cabling which appears at the surface to look similar to TV coaxial cable. It is terminated on each end by a connector that looks much like a phone connector. Its maximum segment length is 100 meters.

6.5.1 Network Categories

There are two main types of network categories which are:

- (a) Server based
- (b) Peer-to-peer

In a server based network, there are computers set up to be primary providers of services such as file service or mail service. The computers providing the service are called servers and the computers that request and use the service are called client computers.

In a peer-to-peer network, various computers on the network can act both as clients and servers. For instance, many Microsoft Windows based computers will allow file and print sharing. These computers can act both as a client and a server and are also referred to as peers. Many networks are combination peer-to-peer and server based networks. The network operating system uses a network data protocol to communicate on the network to other computers. The network operating system supports the applications on that computer. A Network Operating System (NOS) includes Windows NT, Novell Netware, Linux, Unix and others.

6.5.2 Three Network Topologies

The network topology describes the method used to do the physical wiring of the network. The main ones are bus, star, and ring.

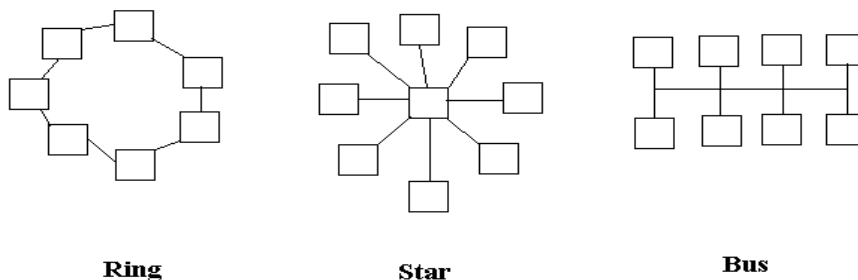


Figure 6.9: Network Topologies

1. Bus - Both ends of the network must be terminated with a terminator. A barrel connector can be used to extend it.
2. Star - All devices revolve around a central hub, which is what controls the network communications, and can communicate with other hubs. Range limits are about 100 meters from the hub.
3. Ring - Devices are connected from one to another, as in a ring. A data token is used to grant permission for each computer to communicate.

There are also hybrid networks including a star-bus hybrid, star-ring network, and mesh networks with connections between various computers on the network. Mesh networks ideally allow each computer to have a direct connection to each of the other computers. The topology this documentation deals with most is star topology since that is what ethernet networks use.

6.6 The Internet

The work Internet emanates from two words. INTER-national and NET-work. Inter - concatenated with Net- becomes Internet.

Internet is the network of network that consists of millions of private and public, academic, business, and government networks of local to global scope that are linked by copper wires, fiber-optic cables, wireless connections, and other technologies.

The Internet carries a vast array of information resources and services, most notably the inter-linked hypertext documents of the World Wide Web (WWW) and the infrastructure to support electronic mail, in addition to popular services such as video on demand, online shopping, online gaming, exchange of information from one-to-many or many-to-many by online chat, online social networking, online publishing, file transfer, file sharing and Voice over Internet Protocol (VoIP) or teleconferencing, telepresence person-to-person communication via voice and video.

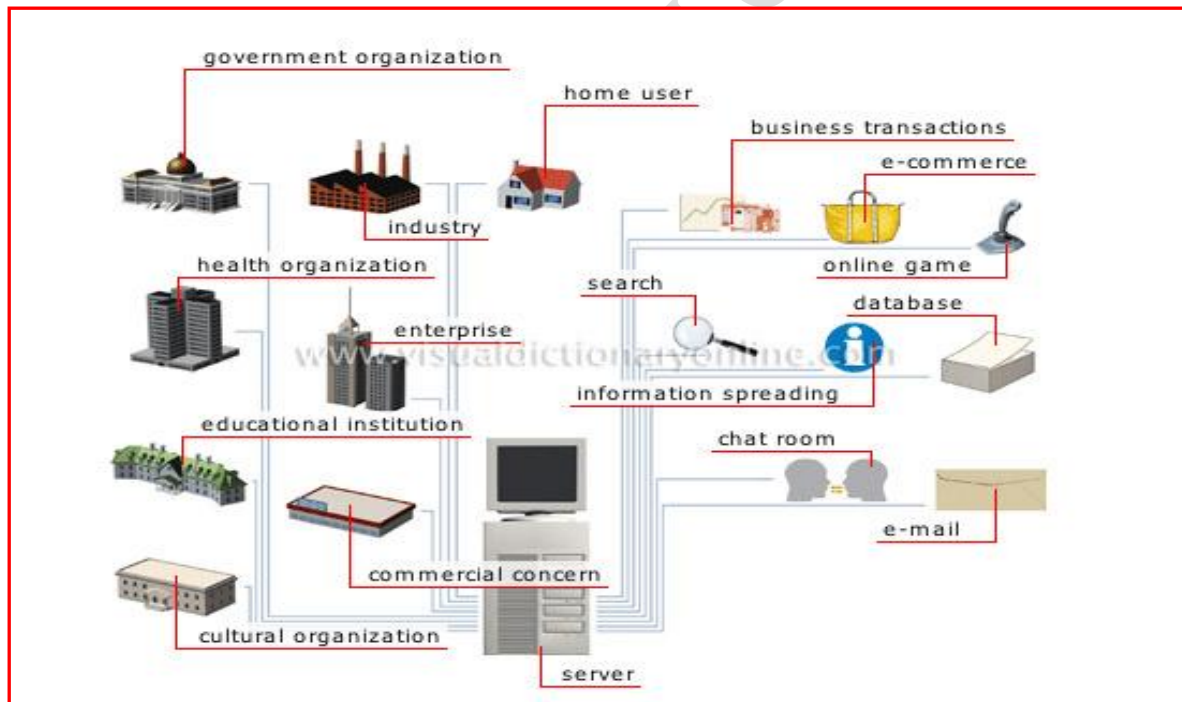


Figure 6.10: Applications of Internet

Internet Browsers

An internet browser is the program that you use to access the internet and view web pages on your computer. Some common internet browser examples include:

- Microsoft Internet Explorer
- Mozilla Firefox
- AOL Explorer
- Apple Safari
- Opera

This is also known as Browser, Web Browser

A browser is a software program that allows you to view and interact with various kinds of Internet resources available on the World Wide Web (WWW). A browser is commonly called a web browser. The most popular web browsers are Microsoft Internet explorer (often called IE for short), Netscape navigator and Mozilla Firefox.

Summary

In this Session, you have been introduced to the concept of Computer Networking. The different equipment that are needed to connect computers together are discussed. The three basic Network Topologies are also explained.

Self-Assessment Questions

1. What is a computer network?
2. Mention any five networking equipment and their functions.
3. Describe the three basic Network Topologies

Study Session 7: Data and Information Representation

Expected Duration: 1 week or 2 contact hours

Introduction

Humans communicate with one another using data consisting of the characters: Alphabets (A to Z), digits (0 to 9), and special symbols such as \$, #, @, ?, etc. Computers are capable to read such characters into primary memory but this data must be converted into a form that permits high-speed internal processing. Computers use the Binary numbering system for representing such characters. The binary numbering system has only two digits, “0” and “1”, each of which is called a binary digit (bit). This system is ideal for computer operation because the bit “1” and “0” are respectively used to denote the presence and absence of electrical pulse or signal in the computer circuitry. In this Session, you will be introduced to how data is represented in computers.

Learning Outcomes

When you have studied this session, you should be able to explain:

- 7.1 Binary Representation
- 7.2 Representing Characters in Storage
- 7.3 EBCDIC Representation
- 7.4 Decimal Equivalence of a Binary Number
- 7.5 Binary Equivalence of a Decimal Number
- 7.6 Remainder Method for Converting Decimal Numbers into any other Base
- 7.7 Binary Arithmetic
- 7.8 Octal and Hexadecimal Number Systems

7.1 Binary Representation

Most numbering systems are called **positional** because the physical location or position of digit within the number affects the value. For example, in a positional numbering system, the number 25 has a different value from the number 52 even though the digits are the same. In positional numbering systems, the place value is critical. Recall that the decimal or base 10 system has the following positional values.

.....	10^3	10^2	10^1	10^0	Exponential values of position
....	1000	100	10	1	Decimal values of position

The binary numbering system has a base of 2. Thus each position has a value that is a multiple of 2. We have then:

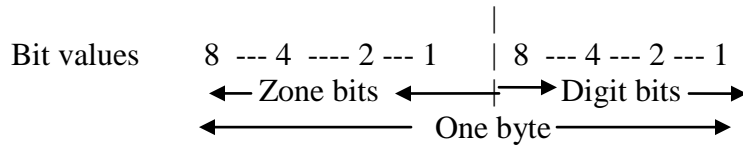
.....	2^4	2^3	2^2	2^1	2^0	Exponential values of position
....	16	8	4	2	1	Decimal values of position

An example of a binary number is 1110

7.2 Representing Characters in Storage

Recall that each storage position in the primary memory is called a byte. If each byte contains four bits, representing the decimal numbers 8-4-2-1, it would be possible to represent any of the decimal digits 0 to 9. That is, with 8-4-2-1 we can represent the decimal digits 0 to 9, as well as numbers 10 to 15. In short, four bits in each byte are used to represent a single decimal digit.

To accommodate alphabets and special characters, the computer frequently uses an 8-bit code with four **zone bits** and four **digit bits**. The four leftmost bits are called zone bits while the rightmost four bits are called digit bits as shown below.

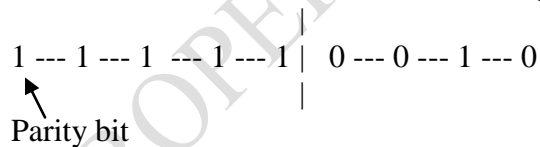


Thus, each byte contains 8 bits; four used for zone representation and four for digit representation.

7.3 EBCDIC Representation

One computer code often used for character representation is called Extended Binary Coded Decimal Interchange Code (EBCDIC). The four zone bits are used to indicate codes for letters, unsigned numbers, positive numbers, negative numbers and special characters. The character set of a computer are grouped into zones such that characters in a zone group has a common zone bits, each character within a zone group has a unique digit bits. For example, 1111 in the zone bits designates a character as an unsigned number; one of the letters A – I. if 1100 appears in the zone bits, the digit bits then will indicate which specific letter from A – I is being represented. Thus, the combination of 4 zone bits and 4 digit bits for a character, gives a unique 8-bit representation for the character.

EBCDIC uses a ninth bit called a **check** or **parity bit** that minimizes the risk of transmission errors. On **even parity computers**, there must always be an even number of bits ON in one storage location at any time. The parity bit is turned ON in order to ensure that there is an even number of bits ON at all times. For example, the decimal digit 2 will be represented as:



So, with EBCDIC each character is represented by a 9-bit code. Other computer codes for character representation include **Binary Coded Decimal (BCD)** and **American Standard Code for Information Interchange (ASCII)**. Some computers use 7-bit ASCII codes, others use 8-bit ASCII code similar to EBCDIC. Table below shows the bit configurations of the EBCDIC, 8-bit ASCII and 7-bit ASCII codes.

Character	EBCDIC	8-bit ASCII	7-bit ASCII
0	1111 0000	0101 0000	011 0000
1	1111 0001	0101 0001	011 0001
2	1111 0010	0101 0010	011 0010
3	1111 0011	0101 0011	011 0011
4	1111 0100	0101 0100	011 0100
5	1111 0101	0101 0101	011 0101
6	1111 0110	0101 0110	011 0110
7	1111 0111	0101 0111	011 0111
8	1111 1000	0101 1000	011 1000
9	1111 1001	0101 1001	011 1001
 			
A	1100 0001	1010 0001	100 0001
B	1100 0010	1010 0010	100 0010
C	1100 0011	1010 0011	100 0011
D	1100 0100	1010 0100	100 0100
E	1100 0101	1010 0101	100 0101
F	1100 0110	1010 0110	100 0110
G	1100 0111	1010 0111	100 0111
H	1100 1000	1010 1000	100 1000
I	1100 1001	1010 1001	100 1001
J	1101 0001	1010 1010	100 1010
K	1101 0010	1010 1011	100 1011
L	1101 0011	1010 1100	100 1100
M	1101 0100	1010 1101	100 1101
N	1101 0101	1010 1110	100 1110
O	1101 0110	1010 1111	100 1111
P	1101 0111	1011 0000	101 0000
Q	1101 1000	1011 0001	101 0001
R	1101 1001	1011 0010	101 0010
S	1110 0010	1011 0011	101 0011
T	1110 0011	1011 0100	101 0100
U	1110 0100	1011 0101	101 0101
V	1110 0101	1011 0110	101 0110
W	1110 0110	1011 0111	101 0111
X	1110 0111	1011 1000	101 1000
Y	1110 1000	1011 1001	101 1001
Z	1110 1001	1011 1010	101 1010

7.4 Decimal Equivalence of a Binary Number

All positional numbering systems have similar structures. To obtain the decimal equivalence of a number in any base, multiply the digits by their positional values and add the results. For example, the decimal equivalence of the binary number 1110 can be obtained thus:

$$\begin{array}{cccc}
8 & \text{---} & 4 & \text{---} & 2 & \text{---} & 1 \\
1 & & 1 & & 1 & & 0 \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
1 \times 8 = 8 & & 1 \times 4 = 4 & & 1 \times 2 = 2 & & 0 \times 1 = 0 \\
\hline
\text{Answer} = & & & & & & 14_{10}
\end{array}$$

Therefore, given any binary number, we can determine its decimal equivalence by the following steps:

1. Determine the positional value of each digit
2. Add the positional values for all positions that contains 1

7.5 Binary Equivalence of a Decimal Number

Numeric data are entered in standard decimal form and then converted by the computer itself to a binary representation. Before the output is produced, it is again converted to decimal form for readability. Using the binary numbering system, the computer can represent any decimal number with a series of bits 1 and 0. How do we determine the binary equivalence of a decimal number?

The decimal-to-binary conversion process is a relatively simple task when small numbers are involved. We merely employ positional values of binary numbers to find the right combination of digits.

Example: To convert decimal number 12 to binary

Remember that the positional values for binary numbers are 1, 2, 4, 8, 16, 32, 64, etc. We need to determine what combination of these positional values will equal 12. It is clear that we do not need to use more than four bits to represent 12_{10} since the fifth positional value is 16, which is greater than 12. Hence, we must determine what combination of 8, 4, 2, 1 will equal 12. There is only one such combination, $8 + 4 = 12$. Thus our binary equivalence is

$$\begin{array}{cccc}
8 & \text{---} & 4 & \text{---} & 2 & \text{---} & 1 \\
1 & & 1 & & 0 & & 0
\end{array}
\text{ That is, } 12_{10} = 1100_2$$

No doubt this method will be cumbersome for larger numbers. Consider using this method for a number like 1087. Therefore we need another method, the **remainder method**, that provides a more convenient method of conversion.

7.6 Remainder Method for Converting Decimal Numbers into any other Base

It takes the following steps:

- (a) Divide the decimal number by the base (for binary equivalence, we divide by 2)
- (b) Indicate the remainder, which will be either 0 or 1 in case of binary division

- (c) Continue dividing into each quotient (result of previous division) until the divide operation produces a zero quotient or result.
- (d) The equivalent number in the base desired is the numeric **remainders** reading from the last division to the first.

Example: Convert 12_{10} to binary.

$\frac{6}{2 \mid 12}$	Remainder	
		0
$\frac{3}{2 \mid 6}$		0
		↑
$\frac{1}{2 \mid 3}$		1
		↑
$\frac{0}{2 \mid 1}$		1

Resultant binary number reads from bottom to top (1100)

7.7 Binary Arithmetic

1. Binary Addition

The addition of binary numbers follows a series of simple rules: for each position beginning with the rightmost:

- $1 + 0 = 1$
- $0 + 1 = 1$
- $0 + 0 = 0$
- $1 + 1 = 0$ with a carry of 1 to the next position

Example 1: $10_2 + 11_2 = (?)_2$

Binary	Decimal
10	2
$+11$	$+3$
101	5

Example 2: $1101_2 + 1010_2 = (?)_2$

Binary	Decimal
1101	13
$+1010$	$+10$
10111	23

2. Binary Subtraction

Binary subtractions can be performed like decimal subtraction. Starting with the digit on the extreme right, each digit of the subtrahend is subtracted from the corresponding digit in the minuend. If the subtrahend digit is larger, then one (1) must be borrowed from the next higher positional value. The result of the complete subtraction is known as the difference.

With computers however, binary subtraction is actually done by **complement addition**. A **complement** of a number is the value which must be added to it to get its number base. For example, in the decimal system (base 10), the 10's complement of 8 is 2, because $8 + 2 = 10$. The 10's complement of 4 is 6, and so on.

Complement addition can be used to attain the same results as with regular subtraction. For example, to perform the arithmetic operation $9 - 3$, we could convert this subtraction to an addition by using the 10's complement of 3, which is 7, and adding it to 9. The result is 16 and any carry is discarded, leaving the answer 6. So $9 - 3 = 9 + 7 = 16 = 6$. So also, $8 - 6 = 8 + 4 = 12 = 2$

You may be wondering why we go through all this? The reason is that a unique property of binary numbers allows the determination of the 2's complement to be very simple, a fact that has important implications for simplifying computer circuit design.

To get the 2's complement of a binary number, all bits in the number are merely switched to their opposite values. That is 0s become 1s and 1s becomes 0s. binary subtraction is performed thus:

- (a) Obtain the 2's complement of the subtrahend by switching all 0s to 1s and all 1s to 0s, and add 1 to the result.
- (b) Add this complement to the minuend
- (c) Discard any extra carry digit in the resulting addition.

Example 1:	Base 10		Base 2
	7 ←	Minuend →	0111
Subtraction	- 3 ←	Subtrahend →	0011
	4 ←	Difference →	0100

Complement Addition:

7	0111
+7 (10's complement)	+1101 (2's complement of 3, $1100+1 = 1101$)
4 (carry digit discarded)	1010
	<u>1 1</u> (carry)
	0100 (carry digit discarded)

What happens if you try to subtract, say 6 from 3, given a negative result, -3? In complement addition, the lack of any carry digit implies a negative number. You simply take the resulting complement addition number, determine its complement, and place a negative sign in front of it.

Example 2: $011 - 1010$ base 2 (i.e. $3 - 6$ base 10)

In decimal, $3 - 6 = 3 + 4 = 7$. There is no carry digit, therefore determine the 10's complement of 7, which is 3. The answer then is -3. The binary complement addition results in 101_2 with no carry digit. The complement of 101_2 is 010 plus $1 = 011$. The answer then is -011 , which is -3 in the decimal system.

This all seems like a roundabout way of performing subtraction, and it is! But the reason for this operation is to simplify circuit design. Instead of having special circuits for addition, others for subtraction, and still others for multiplication and addition, a set of **adder circuits** can be used as the basis for each of these arithmetic operations. As a result, circuit design is less complex and less costly since there are fewer parts. Because the computer can perform operations so quickly, this roundabout approach does not result in excessive time delays.

Binary Multiplication is done by a **Shift-left** and a **add operation**. Notice that in a binary system, a 0 in the multiplier results in 0s in the intermediate results. A multiplication by 1 results in the multiplicand being repeated and shifted to the left. The number of columns to be shifted is a function of what position value the 1 is in. Therefore, binary multiplication can be performed by shifting the multiplicand to the left number of times equal to the position value of the 1 in the multiplier. These shifted results can then be added using regular adder circuits. Verify this approach multiplying 7×12 in binary.

As it turns out, division is the opposite of multiplication. Instead of shifting the multiplier left and adding the intermediate results, **division** is performed by shifting the divisor right and subtracting it from the quotient initially and then from each intermediate remainder. The subtraction would actually be performed using the 2's complement addition method. Try this out too with some numbers and prove that it would work.

Note that higher level mathematical operations are performed by using the basic operations. For example, exponentiation is just a form of multiplication; that is, 2^3 can be performed by multiplying 2 times itself 3 times. Each multiplication can be performed by shifts and by using the adder circuits.

7.8 Octal and Hexadecimal Number Systems

Other useful number systems in computing are the Octal (base 8) and Hexadecimal (base 16) number systems.

The octal number system uses 8 symbols, 0 through 7 as its digits. Note that each of these octal digits can be represented by 3 bits.

The hexadecimal number system uses 16 symbols, 0 through 9 and A through F (The A through F represent 10 through 15), as its digits. Note that each of these hexadecimal digits can be represented by 4 bits.

We can convert from one number system to another using the **successive position value division method** thus:

Start with the highest position value divisor that does not exceed the base 10 dividend number. Divide that number into the dividend. The multiplier is placed in the quotient. The product is subtracted from the dividend. The resulting remainder is then divided by the next lower position value and so on, until the lowest place value is accounted for. The quotient is the equivalent number in the new base number system. Follow this procedure to find the base 2 equivalent of 11 base 10 (11_{10}), 92_{10} and base 8 equivalence of 92_{10} .

Summary

In this session, the following concepts on data representation in computing have been discussed: Binary Representation, Representing Characters in Storage, EBCDIC Representation, Decimal Equivalence of a Binary Number, Binary Equivalence of a Decimal Number, Remainder Method for Converting Decimal Numbers into any other Base, Binary Arithmetic and Octal and Hexadecimal Number Systems.

Self-Assessment Questions

1. Explain how to convert a number in Base two to denary using positional value concept.
2. Convert the following binary digits to base ten: 101101, 100011, 10101101
3. Explain how to multiply two binary numbers together.

Study Session 8: Problem Solving With Computers

Expected Duration: 1 week or 2 contact hours

Introduction

One of the most common misconceptions is that computers are “problem-solving” machines. Nothing could be farther from the truth. The truth is that a computer is useless without a program to control it. A computer works by obeying a sequence of instructions called program. In this Session, you will be introduced to the nature of and guidelines for problem solving in computing.

Learning Outcomes

When you have studied this session, you should be able to explain:

- 8.1 Nature of Computer Problem-Solving
- 8.2 Problem Solving Phases
- 8.3 Some Guidelines for Problem Solving
- 8.4 Algorithm Development
- 8.5 Stepwise Refinement
- 8.6 Programming

8.1 Nature of Computer Problem-Solving

Computer instructions (called programs) are very simple in their effect, the following being typical:

- (a) Read an item of data into store from an input device (an input operation)
- (b) Copy an item of data from one register to another (a store operation)
- (c) Add the contents of two register and place the sum in a third register (arithmetic operation)
- (d) If the content of a register represents a negative number, take the next instruction from a different part of the program; otherwise continue with next instruction in sequence (a logical operation)
- (e) Write an item of data from store to an output device (an output operation)

Anything which can be computed, in principle, can be computed in a finite number of steps by a program consisting of elementary operations such as these:

There are results which are not computable by any machine whatsoever. In these cases, it may be possible to compute an approximation to the desired result. A program to do this is called a heuristic. Heuristics in some sense could be viewed as an approximated solution. Heuristics are also useful when an algorithm exists but is impractically too slow or needs too much store.

A computer does not assist you in organising your thought or inventing a solution procedure. You must have your procedure in hand and the computer merely manipulates data according to

your prescription. If your solution is wrong, the computer output will be wrong. The procedure which you would have the computer try out must specify precisely what to do in all cases. It must be expressed with a level of detail and precision far in excess of that required for normal human communication. Such a solution procedure required by a computer is called an algorithm.

An algorithm is a procedure for solving a specific problem expressed as a finite number of rules which is complete, unambiguous and guaranteed to terminate in a finite number of applications of these rules.

8.2 Problem Solving Phases

Three distinct phases can be identified in problem solving with computers:

- (1) Problem definition: An analysis of the initial situation so as to define the parameters within which the problem must be solved.
- (2) Algorithm design: The formulation of a satisfactory method of acquiring that solution.
- (3) Programming: The implementation and execution of that method of solution, using a programming language.

8.3 Some Guidelines for Problem Solving

1. Understand the problem

Analyse your problem carefully, attempting to understand its specific aspects and the requirements of an acceptable solution. You should have a precise understanding of what your problem really is. Once you can state your problem more accurately, a direction of solution will sometimes become apparent. Understanding a problem is half of the solution. The more complete and precise the understanding, the more apparent may be the course of action.

For problems which primarily manipulate data, decide at the beginning, what data are necessary for the solution and develop every stage of the solution procedure with this decision in mind.

2. Decompose the problem:

As your understanding of a problem becomes refined, your statement of the problem becomes correspondingly more detailed. However, as a problem becomes more detailed and constrained. It also becomes more difficult to deal with all at one time. Therefore, attempt to break your problem into simpler, relatively independent parts, and then focus on the separate parts. In doing this, attempt to break the solution of one sub-problem followed by solution of the next sub-problem, and so on, will provide a solution to the original problem. When focusing on individual parts of your solution, do not completely ignore their context and common goal. If your problem involves performing some

repeated process, attempt to isolate the action which is required in an instance from the repetitive aspect.

When breaking a problem into simpler parts, perform the decomposition gradually in several stages. Make use of more general criteria in the early stages and more specific criteria later. This movement from general to specific considerations is termed **top-down style** of solving problems. Careful, step-by-step, top-down solution of problems will usually lead to almost perfect programs.

3. Review Your Solution

No matter how thorough the analysis, some feature or specific circumstances is almost invariably overlooked when designing the first (second, third, and fourth) version of the solution procedure. Therefore, review and reconsider your proposed solution to ensure that it is complete and correct. Such solution review should be done at every step in the solution. A stitch in time saves nine and a gram of prevention is worth a kilogram of cure.

8.4 Algorithm Development - An Example:

Suppose the problem: Automate a scheme to count the votes in an election between two candidates for one office. Each ballot paper contains two boxes and each voter is supposed to make an X in the box for one candidate of his choice.

Applying the guidelines given in section 8.3 above, solution to this problem can be approached as follow:

1. **Understanding the problem and the required result:**

The desired result is simply a pair of number-the total vote for each of the two candidates. The natural way to obtain these numbers would be to inspect each ballot paper and maintain running totals of validly marked preference. When all the ballot papers have been inspected, the two running totals would represent the desired result.

2. **Decide what data are required for the solution:**

Of course, we need two “variables” with which to maintain the two running totals – “First-total and “Second-total”. In addition, as each ballot paper is encountered; we must also note the content of both boxes. We will let ‘Box-1 and ‘Box-2’ refer to the contents of the two boxes on the ballot-paper currently being examined. These four variables are required.

A variable like ‘First-total’ which is used to keep a running total is called a counter. A counter is repeatedly incremented like the Odometer on a car which counts the total mileage. Such variable has to be initialized with a value, in the case, a zero.

3. Decompose the problem into a sequences of basic steps:

1. Set both counters to zero
2. Tally the votes
3. Display the results
4. End

Obviously step2 needs elaboration. So we have:

2. Repeat (the following) while any unprocessed ballot remains.
 - 2.1 Inspect the next ballot
 - 2.2 Increment appropriate counter
 - 2.2.1 if Box-1 is X then increment First-total
 - 2.2.2 if Box-2 is X then increment Second-total.

We need to refine 2.2.1 and 2.2.2. to take care of invalid ballot. There are two types of invalid ballot:

- X's is in both boxes
- Both boxes are blank

We can then refine 2.2.1. and 2.2.2. as:

- 2.2.1 If Box-1 is X and Box-2 is not X then increment First –total
- 2.2.2 If Box-2 is X and Box-1 is not X then increment Second-total.

4. Review this Solution:

Note that a case where either a ballot contains a mark other than X is not taken care of in this solution. So 2.2.1 and 2.2.2.need further refining to take care of this thus:

2.2.1. If Box -1 is X and Box-2 is blank then increment First-total.

2.2.2. if Box-2 is X and Box-1 is blank then increment Second-total.

Having satisfied ourselves that every possibility has been covered, we now collect all these components into an algorithm.

1. Set both counters to zero
2. Repeat level 2 while any unprocessed ballot remains:
 - 2.1 Inspect the next ballot
 - 2.2.1 if box-1 is X and Box-2 is blank then increment First-total.
 - 2.2.2 if Box-2 is X and Box-1 is blank then increment Second-total.
3. Display results (First-total and Second-total)
4. End

One more-review does not reveal any other omission and so we are ready to convert the algorithm into a complete program using a programming language.

8.5 Stepwise Refinement

The algorithm development approach adopted in the above example is called Stepwise Refinement. The basic idea of Stepwise refinement is this: Given a complex problem, such as developing a program, split it up into several, more or less independent, sub-problems. If we can assume that these sub-problems are solvable we have an outline solution to the original problem. Call this the level-1 description of the solution. Now turn to the sub-problems and solve these separately. If a sub problem is trivial, write down its final solution directly. If not, solve the sub-problem by applying stepwise refinement to it. This generates a set of sub-sub-problems. Assuming that these are solvable, they constitute a second more detailed solution of the original problem. This is the level-2 description of the solution. Go on in this way until all the remaining sub-problems are trivial. Combining all the final solutions of the sub-problems gives the final solution of the original problem in the most convenient form: a complete algorithm design! Splitting up a problem into sub-problems is a refinement step; the successively more detailed descriptions are the levels of refinement.

Of all the skills required to solve a problem using a computer, the ability to develop an algorithmic solution procedure is the most important and unfortunately, the least mechanical. This skill is largely a matter of experience and human insight. Hopefully, the guidelines given above will guide your experience in the right direction. Practice makes perfect is the watchword!

The category of computer personnel that analyses a problem and devises an algorithmic solution to the problem is called a Systems Analyst.

8.6 Programming

1. Programming Language

After one has been satisfied with the algorithm designed for solving a problem, the next step is to re-write the algorithm in a language which the computer is able to interpret. English is too imprecise; the meanings of its words depend too much on nuance (tone), context, and human experience. Algorithms must be expressed in a programming language having only a limited vocabulary and extremely precise, grammatical rules for building unambiguous instructions from the basic vocabulary. Thus learning to use a computer also requires learning some “foreign” programming language. Examples of programming languages are BASIC, FORTRAN, COBOL, PASCAL, JAVA, etc. An appropriate choice of programming language has to be made.

2. Program Coding

When an algorithm is expressed in a programming language, it is usually called a program. Programs are usually written into an editor of a computer such as Notepad, WordPad, or specialized text editors such as EditPlus. Nowadays, most of the translators come with an Integrated Development Environment (IDE) where codes can be written, debugged and compiled.

3. **Program Compilation:**

A source program is translated into an object program by a translator called the compiler or interpreter. The print out of a source program by the computer on a printer form is called Source Listing.

4. **Program Linking/Loading**

An object program is transformed into an executable program by a program Linker/Loader. Executable program is the source program in the form it can be executed by the computer. At this linking/loading stage, program modules in the system library (if any) that are necessary in executing the program are added to the program.

5. **Program Execution**

The computer executes the executable program with the relevant input data. Once a program is prepared correctly, it can be executed over and over again, applying the same program to different data each time. Any time a program is executed on the computer, we say that we are running the program. A unit of program to be run in a computer is often called a Job.

6. **Program Testing and Debugging**

Program execution is not the final step. It is always necessary to test a program to ascertain whether or not it is able to produce correct results. If the results are not acceptable, then some part of the problem solving process must have contained some errors (bugs), which need to be corrected. The process of correcting errors is called program debugging and it is easily the most gruesome and frustrating of these steps. Unfortunately, far too many programmers spend the major portion of their time debugging their proposed solution. If more time and attention were devoted to initial program preparation – and to general problem solving in particular – there would be fewer errors to eliminate.

The period when a compiler is translating a source program into an object program is called **compile time**, while the period when the computer is executing the program is called **execution time or run time**. Similarly, errors in program encountered and reported by the compiler at compile time are called **compile time errors**. Compile time errors are mostly errors dealing with the violation of the syntax rules of the programming language. **Run time errors** are mostly errors dealing with the incorrect program logic, violation of some semantic rule of the programming language, e.g. violation of some mathematical rule in evaluation of an expression, violation of the rule for data representation in the computer etc.

7. **Programming personnel:**

Those who write computer programs are called programmers. There are categories of programmers: (1) Application programmers, (2) Systems programmers and (3) Maintenance programmers. Application programmers write programs for specific application problems e.g. writing a program for solving simultaneous equations. Systems programmers develop systems programs e.g. operating systems, translator, etc. Maintenance programmers modify existing programs to make them more current or more efficient. In a small or medium-sized computer installation, the same person could serve both as systems analyst and programmer,

or different persons may serve the different roles, depending on the scale of operation of the installation.

Summary

In this Session, the following concepts of Problem Solving with computers were explained

- (a) Nature of Computer Problem-Solving
- (b) Problem Solving Phases
- (c) Some Guidelines for Problem Solving
- (d) Algorithm Development - An Example:
- (e) Stepwise Refinement
- (f) Programming

Self-Assessment Questions

Using the Quadratic Equation as a guide, discuss the principle of problem-solving with computers.

PROPERTIES OF DLC UI, IBADAN

Study Session 9: Elements of Computer Programming I

Expected Duration: 1 week or 2 contact hours

Introduction

This Session will gradually introduce you to the basic elements of Computer Programming. By the way, Programming is one of the bedrocks of Computer Science. Knowledge of programming is very crucial in computing. The Session begins at defining programming and elements or basic concepts of programming are later discussed.

Learning Outcomes

When you have studied this session, you should be able to explain:

- 9.1 The Meaning of Programming?
- 9.2 The Constituents of Programming Environment
- 9.3 Basic Programming Elements
- 9.4 Form of a programming Language
- 9.5 Structure of a Program

9.1 What is Programming?

Before we define the term programming, let us firstly define the term program. A program is a set or sequence of instructions written and compiled in a specific programming language given to a computer to perform a specified task. Therefore programming is the art and science of creating (developing) computer programs. It is the process of writing, testing, debugging/troubleshooting, and maintaining the source of code of computer programs (Wikipedia, 2007). A software is made up of several independent programs working together to perform some tasks. For instance, an operating system is a system software having several functional modules (programs) working to achieve the goal of coordinating the entire hardware resources.

Generally in programming, we perform the following fundamental tasks: Input – Process – Output. Therefore, in learning a programming language's syntax and semantics, we look for the commands or statements that we will use to perform inputs, i.e., how to read in data into the computer via the keyboard, files, databases or any other source. We then learn the commands or statements to process the data, i.e., the process logic. Finally, we learn how to display our results as outputs to users of our programs. That is, how to display via the monitor, store results into a file or database and/ or send the results to other computing system(s).

9.2 Programming Environment

The programming environment includes all the facilities and tools, which a programmer requires in the design, development, testing, implementation and maintenance of a system of programs. Thus, text editors, compilers, interpreters, diagnostic tools, optimizers, measurement tools and other aids comprise a complete programming environment. Apart from the tools mentioned as comprising the programming environment, the following are additional tools needed in programming:

1. A previous knowledge of the pros and cons of programming languages syntax and semantics;
2. Logical reasoning ability; and

3. Algorithms and flow charts.

For a programmer to be successful, he must have a good reasoning ability and previous knowledge of any suitable language that must be used in solving a problem; because any problem that cannot be used by human, computer too cannot solve it. The common maxim in programming is “garbage in garbage out”.

9.3 Basic Programming Elements

When you want to learn a programming language, what things do you expect to see embodied in the language? A typical programming language would embody the following elements:

1. Data and Its Description:

The purpose of a program is to accomplish some types of computation; where computation is not limited to numerical calculations. The elements on which the computation is to be performed are called data. Data might consist of numeric quantities (e.g. 50958, alphabetic quantities, strings of characters or anything else permitted by the language. The data might even be generated completely internally from the program. In most cases, there arises the need for the concept of data variables whose values are to be determined during the execution of the program. Because of the multiplicity of data types which can be used, there is a need for descriptions of them. The methods of describing data vary considerably, ranging from implicit assumptions to specific declarations.

Data description enables the system to know the data on which it is working and takes the form often called **declarations**. Data declarations enable data elements to be given names called **variables** or **identifiers**, and their attributes, (e.g. types, storage requirements) to be specified in the program. Declarations do not cause action to be taken by the computer at execution time, rather they are directives to the compiler to make some information available to the computer about the data on which it is going to operate. A declaration could be made implicit or explicit.

2. Operators

The use of operators is one of the ways of combining or acting on data elements. Operators generally fall into **computational**, **relational**, or **logical** category. Operators generally appear in expressions and do not themselves necessarily cause permanent results (e.g. writing If $A = B + C$ does not create result $B + C$). The common **computational operators are addition, subtraction, division, multiplication, and exponentiation**. These can be represented by any symbols chosen by the language designers, including specific words. Thus, one language might permit the user to write $A + B$, while another requires $A \text{ PLUS } B$. **The relational operators, e.g. GREATER THAN, EQUAL TO, LESS THAN, $<$, $>$, $<=$, $>=$, and varying combinations of these, are commonly used to compare arithmetic quantities but the results is (at least implicit) a logical value. Common operators for logical data (data which can have only the values TRUE and FALSE) are AND, OR, NOT.**

3. Command Statements

The heart of a language is the set of executable actions that can be performed on the data elements. Each command performs a specific task as specified by the language designers. Command statements in a programming language would include:

- Assignment statements (e.g. assign a new value to a variable)
- Sequence control (e.g. transfer control to another command)
- Input and Output statements (for data input and result output).

4. Delimiters

Delimiters are a part of the language which serves only the syntactic purpose of helping to define various other parts of the language. A delimiter can be any combination of tokens that the language designers feel desirable. For example, a delimiter can be a key word, a particular punctuation symbol, blank or varying combinations of these. The prime purpose of the delimiter is to define the beginning and /or end of elements in the language.

5. Program Structure

Assuming that a language contains the elements discussed above, there must be a meaningful way of combining these to produce some desired action. The way in which this is done is called the program structure. This concept involves the rules needed for combining sets of commands and the data on which they operate. It also provides rules for building larger program from smaller ones. This is discussed more in a further section below.

9.4 Form of a programming Language

The form of a language can be considered to consist of the character set, the basic elements (token) and definition and usage of other basic elements.

1. Character Set:

The fundamental constituent of a programming language is the character set which it uses. Character sets allowable are obviously constrained by the hardware available and, as a result, the most common classes are those which use the 47 (or 48) characters of the key punch machine and those that use the characters of the key punch machine and those that use the character on a typewriter. Most computers and hence, programming languages, support ASCII (American Standard Code for Information Interchange) character set, which include the following in addition to the letters and digits.

! @ # \$ % ^ & * () - + _ = { } [] : ; “ ” ‘ ’ | / \ blank

2. Basic Elements (Tokens):

The word Token is used to refer to the basic elements in a language. In this context, the elements are atomic, i.e., they have no possible further subdivisions. A token may be a single character, or a sequence of characters surrounded by spaces. A token may be user-defined or system defined. System-defined tokens are operators, key words, and punctuation symbols. User defined (or more precisely, created in his program) tokens include identifiers, constants, literals and comments.

For any program, the concepts of data and variable exist in some form. Data, however, must be able to be referred to in some general way. This is done by given it a name, and the name is more rigorously called an identifier.

Similarly the concept of variable – i.e., a quantity whose value changes during the program execution – exists, and it must be named or identified. There is a significant difference between an identifier and the item it is naming. The identifier may refer primarily to a storage location to a whole group of data elements or a formal variable which never receives any value. An identifier can be a data name or a program unit name. Identifiers are formed according to specific formation rules laid down by the language designer. For instance, such rules may include the statements that, key words in a language are often termed reserved words which cannot be used as identifiers. There may be a maximum limit to the number of characters to form an identifier; an identifier may always have a letter as its first character.

Further, most programs require the use of some fixed quantities during the course of the computation. The quantities are most usually members, or logical or character string or constants. A constant is one of the user-defined basic elements in a programming language.

A special type of constant is known as Literal. A number of cases arise in which one wishes to use the string ABC to mean an identifier. On the other hand there may be times when one wishes to use the string ABC to mean exactly itself. In this latter usage we call the string ABC, a literal. In other words, a literal is a string of characters which represents itself and not something else.

Since one of the advantages of a high-level programming language is to provide better documentation of the task being performed, it is essential that there be a means of providing “comments” in the program. Comments are one of the possible types of user-defined tokens. Most programming languages provide a method by which the user intersperse comments into his program.

9.5 Structure of a Program

Merely a long list of syntactically correct character sequence is not going to necessarily produce a meaningful program, and all languages have types of subunits which must combine properly to make a complete program. The following such subunits can be recognized in a language:

1. **Non-executable**

All compiler directives including declarations and also comments are non-executable subunits of a program.

2. **Smallest Executable Unit:**

The smallest executable unit (SEU) is a general name for what is usually a single command and its operands. e.g. the assignment statements $Y = 3$ and the output statement `PRINT Y`, are executable units.

3. **Blocks or Compound Statement:**

A set of SEU's can be grouped together to form a unit called a Block or Compound statement. A block normally has some kind of designator to indicate its beginning and end; the words `BEGIN AND END` or `{ and }` are, in fact, often used.

4. **Loops:**

A common and essential part of a programming language is the capability of repeating a certain sequence of executable units for more than one value of a particular parameter (or set of parameters). This concept is called a loop. A loop has four constituents: Range, the value(s) of the parameter(s), the terminating condition for the loop and the place to which control is transferred when the execution of the loop is finished.

5. **Functions, Subroutines and Procedures:**

A subroutine is simply a self-contained set of statements to perform a particular task e.g. a subroutine could be developed for such commonly needed computations as, finding $\text{Sine } x$ or finding roots of polynomial, or inverting matrix. Most large programs are built up from a series of subroutines. The special case of a subroutine which has a single result is usually called a **Function**, otherwise it is called a **Procedure**. The definition of a subroutine is usually written as a self-contained unit called the body, outside the main control flow of the program and invoked from within the program. The process of invoking requires supplying parameters, unless the subroutine does not allow them.

Summary

In this Session, you have been introduced to the meaning of computer programming, programming environment, elements and structure of a program.

Self-Assessment Questions

1. What is programming?
2. Explain the structure of a program.

Study Session 10: Elements of Computer Programming II

Expected Duration: 1 week or 2 contact hours

Introduction

Good, logical programming is developed through good pre-code planning and organization. This is assisted by the use of algorithms, pseudocodes and program flowcharts. Some control structures are also needed to organize your programs in order to achieve your goals. In this session, you shall be introduced to program designing with algorithms, pseudocodes and /or flowcharts. The Control structures are also explained

Learning Outcomes

When you have studied this session, you should be able to explain:

- 10.1 the meaning and use of Algorithms
- 10.2 the meaning and use of Pseudocode
- 10.3 the meaning and use of Flowcharts
- 10.4 the meaning and use of Branches
- 10.5 the meaning and use of Loops

10.1 Algorithms

The word "algorithm" comes from the Latin form of the name Al-Khwārizmī, a Persian mathematician who lived in the 8th century AD. An algorithm is the step-by-step solution to a certain problem. Algorithms are a series of step by step instructions for solving a problem. Algorithms are lists of instructions which are followed step by step.

An algorithm literarily means the step-by-step procedure of solving a problem. Technically, it is defined as a finite solution steps to problem. As a matter of fact, an algorithm must have the following properties:

- (i) **Finiteness:** It must have a terminating point after some points.
- (ii) **Definiteness:** It must be clear and unambiguous.
- (iii) **Input:** It must have allowable set of data input to the system.
- (iv) **Output:** It must properly define the nature of result expected; that is, the format.
- (v) **Efficiency:** Its implementation must be space (memory) and time efficient.

As an illustration, suppose we want to add some numbers together continuously until when we encounter any negative data, which shall terminate the addition. In addition, we also want to report the total number of valid data added so far before the program halts.

The algorithm:

1. Set up a counter, count = 0
2. Initialize an accumulator, sum = 0
3. Read in data, d
4. If ($d \geq 0$) then
 - Sum = sum + d
 - Count = count + 1
 - Goto 3

Else
Print out Results, Sum, Count
Terminate

10.2 Pseudocodes

Pseudocode is a method of describing computer algorithms using a combination of natural language and programming language. It is an algorithm written in a way that resembles computer code. It is essentially an intermittent step towards the development of the actual code. It allows the programmer to formulate their thoughts on the organization and sequence of a computer algorithm without the need for actually following the exact coding syntax. Although pseudocode is frequently used there are no set of rules for its exact implementation. In general, here are some rules that are frequently followed when writing pseudocode:

- (a) The usual programming symbols are used for arithmetic operations (e.g. +, -, *, /, **).
- (b) Symbolic names are used to indicate the quantities being processed.
- (c) Certain language keywords can be used, such as PRINT, WRITE, READ, etc.
- (d) Indentation should be used to indicate branches and loops of instruction.

To practice writing algorithms in pseudocode, you first think of a complex problem and consider the logical solution before writing it out. Pseudocode (and programs in general) are best suited for mathematical problems.

10.3 Flowcharts

Flowcharts are basic symbolic representations of solution pathway to a problem. In other words, it is diagrammatic representation of solution to problems. Flowcharts are graphic means of describing a sequence of operations done on data. They serve as a means of communication from one person to another about transformations on data. Flowcharts are graphic because they commonly use two-dimensional pictorial format. Flowcharts get their names from their chart (graphic) representation of the flow (orderly passing of control) from one operation to the next in an explicit sequence.

Flowcharts go by many other names, including block diagram, process chart, procedure chart, and logic chart. These different names reflect in part a lack of uniformity of nomenclature and in part the particular interests of specialized users. For example, prior to the advent of computers the name “flowchart” was used by systems analysts to designate a means of describing the flow of documents carrying data in an organization. Flowcharts are the most widely used graphic method for describing computer operations. They are adaptable to a wide variety of different applications and circumstances and have enjoyed use from the early days of the computer field.

10.3.1 Use

The major use of flowcharts is in documentation and in programming. As a documentation device, the flowchart provides a way of communicating, from one person to another, the nature of the operation to be performed and of the data upon which it is to be performed, regardless of the programming language or computer used. Since a flowchart is a graphic means of

communication, this feature makes it a good choice for use with the usual programming language and English language descriptions included in most documentation of computer programs.

As a programming aid, flowcharts are often prepared by systems analysts and designers to describe systems and to specify the work to be accomplished by programs. Programmers use flowcharts as a basis for writing programs and as a means of communicating among each other, particularly when the programming is done as a team effort. Programmers as well as systems analysts also use flowcharts as a source of information for maintenance work on programs and system.

10.3.2 System Chart and Flow Diagram

The two major varieties of flowchart in present-day practice are the system flowchart and the flow diagram (or Program Flow Chart). In a system chart the unit of data transformation is usually the work done by an entire computer program. E.g., sorting a file of data, inverting a matrix, or producing a report. A system flowchart identifies major sets or files of input and output data handled by the programs, people and machines involved in a system. It is commonly prepared from the specialized outlines used to indicate the media or the equipment employed in the system to handle the data.

By contrast, in a flow diagram (or a program flow chart), the unit of data transformation is usually an operation or short sequence of operations that a computer can perform, such as an instruction or a series of instructions that comprise a subroutine. A flow diagram describes algorithms, usually as they are implemented in a computer program. This may be a reason why it is also called a Program flowchart.

10.3.3 Flowchart Elements

Wide agreement exists on the major elements of flowcharts. They are the symbols and flow representation (sequence). The symbols are in three groups: the basic, the specialized and others.

Complete flowcharts can be drawn using only the basic symbols. These symbols, as shown below are a parallelogram for input and output, a rectangle for processing and a line or line with open arrowheads to represent the direction of flow. When only basic symbols are used, the rectangle serves for all operations except input and output. To provide explanatory comment, the annotation symbol (a partial rectangle attached by a dashed line) can be used, with the comment written within the rectangular part.


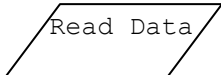
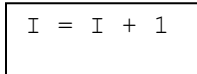
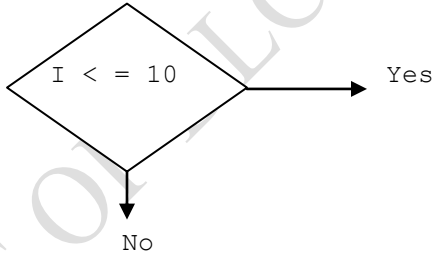


Specialized symbols offer a way of visually identifying for the user of the flowchart the media used to carry data: the equipment used for input, output or storage of data; and the general character of a processing operation such as data transformation.

In system charts, the most commonly used of these specialized symbols are usually document, magnetic tape, on-line storage (for data on magnetic disk), and punched card symbols. In flow diagrams, the most commonly used are the preparation, the decision and the predefined process. A predefined process is considered to be a sequence of operations not described in the flowchart but incorporated in the program, as by a call to a subroutine in a library. The manual operation symbol is for representing operations done by people, such as “review of data for conformity to

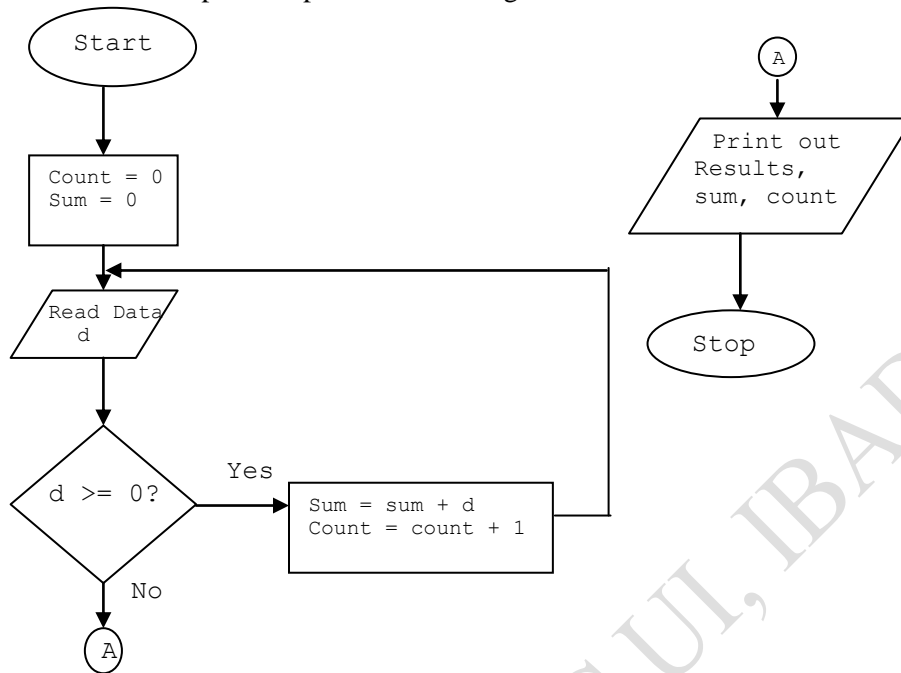
policy". The auxiliary operation symbol is for representing operations done by non-computer machines, such as interpret punched card" done in those days.

The other additional symbols are connectors of three types, and serve to indicate that two or more sequences are to be performed simultaneously (parallel mode). The in-connector and the out-connector are distinguished by the flow lines leading to and from them. A terminal connector may also be used in an entrance or exit position as a marker to indicate the beginning or ending of a sequence of operations.

The following basic symbols are common used:

1. Oval Shape for start or stop 
2. Parallelogram for Input and output 
3. Rectangle for process 
4. Diamond shape for Decision taking 
5. Arrow for Flow or direction 
6. Small Circle for connecting two parts of the flowchart together 

The Flowchart for the above specified problem under algorithm is shown below.



The flowcharting conventions can be generally summarized thus:

- (i) Each symbol denotes a type of operation.
- (ii) A note is written inside each symbol to indicate the specific function to be performed
- (iii) The symbols are connected by flow lines; flowcharts are read from top to bottom and left to right.
- (iv) A sequence of operations is performed until a terminal symbol designates the end of the run, or a branch connector transfer's control to another part of the flowchart.

Even though the flowcharts are no more recommended as part of programming tools nowadays, it is still a starting point for new programmers. We shall see the usage of flowchart very soon.

Here is another example problem, including a flowchart and pseudocode/algorithm. This problem and solution are from Nyhoff, pg 206:

For a given value, *Limit*, what is the smallest positive integer *Number* for which the sum $Sum = 1 + 2 + \dots + Number$ is greater than *Limit*. What is the value for this *Sum*?

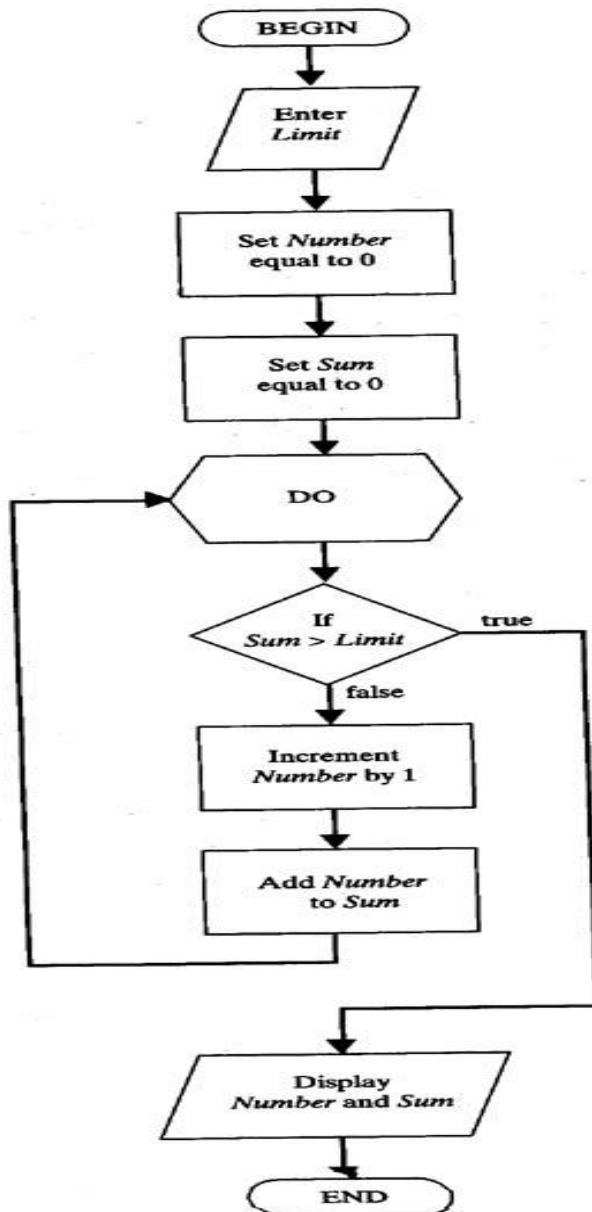
Pseudocode/Algorithm:

Input: An integer *Limit*
 Output: Two integers: *Number* and *Sum*

1. Enter *Limit*
2. Set *Number* = 0.

3. Set $Sum = 0$.
4. Repeat the following:
 - a. If $Sum > Limit$, terminate the repetition, otherwise.
 - b. Increment $Number$ by one.
 - c. Add $Number$ to Sum and set equal to Sum .
5. Print $Number$ and Sum .

The Flowchart:



10.4 Branches

Another important technique when writing program code is the ability to select different paths of execution. Your code may need to branch depending on different criteria. The most important of these are:

- IF (IF-THEN)
- IF-ELSE
- IF-ELSE IF
- SELECT CASE

10.4.1 IF Control

The simplest form is the logical IF statement. In general:

IF (*logical -criteria*) execution statement

If the *logical-criteria* is TRUE then the execution is performed. If not the execution statement is bypassed. An example:

```
IF (2.0 < x .AND x < 3.0)
  PRINT x
```

Other statements not affected by the IF follow from here.

In this case if x is between 2 and 3 then it will be printed. If the condition is not true, the control jumps to other statements.

Another IF form used when multiple statements are required for the TRUE case is:

```
IF (logical -criteria) THEN
  execution statements
END IF
```

Other statements not affected by the IF follow from here.

This is the block IF statement where all the affected statements are enclosed within the THEN and END IF keywords. In some modern languages such as C, C++, C# and Java, the THEN is an opening curly brace while the END IF is replaced by the closing curly brace.

```
IF (logical -criteria) {
  execution statements
}
```

Other statements not affected by the IF follow from here.

Notice that all execution statements depending on the IF statement are indented. This is not required, but it makes your code more readable. All the execution statements are performed if

the logical statement is TRUE. Otherwise, all information between the IF and END IF statements are ignored. An example:

<pre>IF (x >= 0) THEN z = x * y PRINT "x is a positive number." END IF</pre>		<pre>IF (x >= 0) { z = x * y PRINT "x is a positive number." }</pre>
---	--	---

10.4.2 IF-ELSE Control

The ELSE statement allows specification of execution statements for the case that the *logical-criteria* of the IF statement is FALSE.

<pre>IF (<i>logical -criteria</i>) THEN execution statements for true result ELSE execution statements for false result END IF</pre>	or	<pre>IF (<i>logical -criteria</i>) { execution statements for true result } ELSE { execution statements for true result }</pre>
--	----	---

Other statements not affected by the IF - ELSE follow from here.

If the *logical-criteria* is TRUE then the first set of statements are performed and the second are bypassed. If the *logical-criteria* is FALSE then the first set of statements are bypassed and the second statements performed. As an example:

<pre>IF (x > 0) THEN PRINT "The value is greater than zero." ELSE PRINT "The value is not greater than zero." END IF</pre>		<pre>IF (x > 0) { PRINT "The value is greater than zero." } ELSE { PRINT "The value is not greater than zero." }</pre>
---	--	---

10.4.3 IF-ELSE IF Control

Nested IF statements (IF statements within IF statements) can allow for many selection criteria. For example:

```
IF (x >= 1) THEN
  PRINT "The value is greater than or equal to one."
ELSE
  IF (x > 0) THEN
    PRINT "The value is between zero and one."
  ELSE
    PRINT "The value is less than or equal to zero."
```

```
END IF
END IF
```

Although nested loops are allowed, they can frequently become very complex if many selection options are desired. **The ELSE IF statement allows for multi-alternative selection in a more simple and easier to read format.** In general:

```
IF (logical -criteria1) THEN
    execution statements 1
ELSE IF (logical -criteria2) THEN
    execution statements 2
ELSE IF (logical -criteria3) THEN
    execution statements 3
....
ELSE
    execution statements n
END IF
```

The ELSE statement is optional, but it indicates the statements to be performed if none of the IF statements are TRUE. Only one set of execution statements are performed. The expressions are evaluated in order and when a TRUE statement is found, its execution statements will be performed and execution continues with the next statement following the END IF statement. For example:

```
IF (x > 0) THEN
    PRINT "Value is greater than zero."
ELSE IF (x < 0) THEN
    PRINT "Value is less than zero."
ELSE IF (x == 1) THEN
    PRINT "Value is one."
ELSE
    PRINT "Value is zero."
END IF
```

Notice if x is equal to 1, the output of this series of commands will be:

Value is greater than zero.

It will not be "Value is one." because the first true IF statement is that it is less than zero. That statement was executed and execution proceeded outside of the IF-ELSE IF construct.

10.4.4 SELECT CASE

The SELECT CASE construct is not as general as the IF constructs, but it is of use for some specific applications. In general:

```
SELECT CASE (selector)
  CASE (list1)
    execution statements 1
  CASE (list2)
    execution statements 2
  ...
  CASE (listn)
    execution statements n
END SELECT
```

The *selector* is either an integer, character or logical expression. **It cannot be real.** *List1* to *listn* are the possible values for the *selector*. Say you wanted to associate output a letter grade for a known numerical score. Say the numerical score is associated with the real variable named grade:

```
SELECT CASE (INT(grade)) ! The real value grade is converted to an integer.
  CASE (90:)             ! 90: indicates values of 90 or above.
    PRINT "Your grade is an A."
  CASE (80:89)           ! 80:89 means 80 to 89.
    PRINT "Your grade is a B."
  CASE (70:79)
    PRINT "Your grade is a C."
  CASE (60:69)
    PRINT "Your grade is a D."
  CASE (:59)             ! 59: indicates 59 or below.
    PRINT "Your grade is an F."
END SELECT
Other Statements follow
```

10.5 Loops

Often it is useful to have repeated execution of a particular set of Program expressions. This can often be accomplished using the DO or FOR command depending on the language. There are various ways for loops to function, but in general they are either based on counters where loops are continued for a certain number of iterations, or logical expressions where the loop terminates based on the value of a logical expression.

10.5.1 Counter Loops

Counter loops perform repetitive execution a number of times based on a control-variable. The control-variable is initialized to a value and incremented a step-size every time a particular list of

commands is executed. When the control variable surpasses a specified threshold the execution halts. The general form of the counter controlled DO or FOR loop is:

DO <i>control-variable</i> = <i>initial value, limit, step-size</i>	FOR I = Initial value To Final STEP k {
execution list	execution list
END DO	}

Initial-value, *limit*, and *step-size* are all integer values that indicate the number of loops to be performed. These values can also be variables indicated previous to execution of the loop. *Step-size* is not required. If not indicated, the default *step-size* is one. The values for *initial-value*, *limit*, and *step-size* are indicated at the beginning of the loop by either numerical values of variables. These values cannot be changed during execution of the loop.

The execution list within a DO / FOR loop should be indented. This makes your code more readable, and it is easier to follow the logic. Often you will have DO loops within DO loops (nested DO loops), and in these cases **indenting each new loop within the other loops allows you to follow the sequence of loop execution.** Here is an example of a counter DO/FOR LOOP.

max = 5	max = 5
DO n=1, max	FOR N = 1, max {
PRINT n, n*2	PRINT n, n * 2
END DO	}

This output:

```
1  2
2  4
3  6
4  8
5 10
```

Step-size was not indicated; so, it is one. The control-variable cannot be modified in the execution section of the loop, but it can be used in the execution statements. The following DO loop shows a loop nested within another loop:

```
DO n = 5, 1, -1 ! From 5 to 1 with a step size of minus 1
  DO m = 1, 2
    PRINT n, m
  END DO
END DO
```

Notice how by indenting the loops the order of execution is indicated. The most indented loop is performed in completion first. The output would be:

```

5 1
5 2
4 1
4 2
3 1
3 2
2 1
2 2
1 1
1 2

```

Loops can be labeled as another way to organize your nested loops. The previous nested loops could have been written:

```

outer: DO n = 5, 1, -1 ! From 5 to 1 with a step size of minus 1
      inner: DO m = 1,2
            PRINT *, n, m
          END DO inner
        END DO outer

```

If you have many nested loops this could assist in helping you and others follow your code.

10.5.2 logical loops

Logical loops are loops that require fulfilling a logical criteria for completion of the loop. These can also be called DO-EXIT loops. The general form is:

<pre> DO execution statements IF (logical criteria) EXIT execution statements END DO </pre>	<pre> FOR (; ;) { // Starting an infinite loop Execution statements IF (logical criteria) break execution statements } </pre>
---	---

The execution statements can be before and/or after the EXIT statement. When the IF statement is TRUE, repetition is halted. Be careful when determining the *logical criteria* for your IF statement. **If it never becomes TRUE you will be left with an infinite loop.**

The most common use for a logical loop involves halting execution once a calculated value reaches a threshold. For example:

<pre> n = 1 DO IF (n > 10) EXIT n = n + 1 END DO </pre>	<pre> n = 1 FOR (; ;) { IF (n > 10) break n = n + 1 } </pre>
--	---

You can also use user input to halt repetition. For example:

```
pi = 3.142
DO
  PRINT "Enter the radius of your circle."
  READ r
  PRINT "Area is ", pi*r**2
  PRINT, "Do you want to calculate another area?"
  READ response
  IF (response == "n") EXIT
END DO
```

Granted, the parameter pi must have been defined previously and response would need to be defined as a character variable of length = 1 in the specification section of the program, but you see how the user input was used as a decision criteria.

A more modern and structured way of realizing a logical loop is with use of the DO WHILE - END DO expression:

```
DO WHILE expression
  execution statements
END DO
```

Here the execution statements are performed as long as *expression* returns TRUE. When *expression* becomes FALSE the loop is broken and execution continues with the following statements. The equivalence between DO-EXIT and DO WHILE loops is portrayed below:

```
n = 1
DO
  IF (n > 10) EXIT
  n = n + 1
END DO
```

← these two command list are equivalent →

```
n = 1
DO WHILE (n <= 10)
  n = n + 1
END DO
```

Summary

No doubt this Session has further introduced to other elements of computer programming. The different concepts learnt, such as Algorithms, Pseudocode, Flowcharts, Branches and Loops would help you to develop meaningful and useful programs.

Self-Assessment Questions

1. Explain the differences among programs, pseudocode, flowchart and algorithm.
2. Design an algorithm to compute the roots of any quadratic equation
3. Explain the structures and uses of the following control statements
 - (i) If – Else
 - (ii) For – loop
 - (iii) Do - While

Study Session 11: Programming With Python Language

Expected Duration: 1 week or 2 contact hours

Introduction

Python is an object oriented programming language. Python is easy to learn and also easy to develop programs with graphics, sound capabilities for games, animation for Arts and sciences and output programs for the web (CGI). It is named after a British comedy called Monty Python. This Session and others following will introduce you to Python Programming Language.

Learning Outcomes

When you have studied this session, you should be able to explain:

- 11.1 The reasons for Python programming language
- 11.2 How to Download and install Python
- 11.3 the features of Python Environment
- 11.4 How to start the IDLE Python Interpreter
- 11.5 How to use Files
- 11.6 Python Variables
- 11.7 Identifiers
- 11.8 Reserved Words/Keywords
- 11.9 Named Constants
- 11.10 How to display the Contents of Variables and Constants: Output

11.1 Why Python?

Python is a modern programming language that was developed by Guido van Rossum (now at Google) in 1990 and first released in 1991. While there are a number of programming languages that might be used for a first course in programming, Python offers a number of features that make it particularly applicable:

- (i) It is interpreted, which for the introductory student means that they can type program commands into a console and immediately see the results. This feature makes learning details of the language much easier.
- (ii) The syntax of Python is generally simpler than other languages. Python's philosophy is "one way to do it" so that the number of details the student has to learn and remember is reduced. This philosophy makes code more natural and more readable. The focus of learning code development should be on its readability, and Python supports it.
- (iii) Python provides high-level data structures and methods that make many programming tasks easier. This feature means that students are more productive: writing significant code, much more quickly.
- (iv) Python has many libraries that make tasks easier. In particular, there are many libraries specific to a field (biology, chemistry, physics, etc.) that make the language useful for

“getting work done.” Therefore, a student who knows Python has a host of software available to them for doing anything from graphics, to gene structure matching, to financial accounting, to music and anything in between.

- (v) Python is free! Students can download it from the web for their personal machines.
- (vi) Furthermore, Python does not depend on a particular operating system. Thus students are free to develop their code on Windows, Macintosh OS X or Linux and the code will (for the most part) run anywhere.

Other important characteristics of Python are

- Python is simple
- Python is object-oriented
- Python is interpreted
- Python is robust
- Python is architecture-neutral
- Python is portable
- Python is multithreaded
- Can be applied in all fields (arts and sciences)

11.2 Downloading Python

Python is available for any platform (Windows, Mac, Linux) for free download. The main Python site is <http://www.python.org> and from there you can download a full installation of Python. Installation instructions are available at the web site as well. You can download Python (version 3.X) from <http://www.python.org/download/>. Note X can be any number depending on the latest version you download from the site.

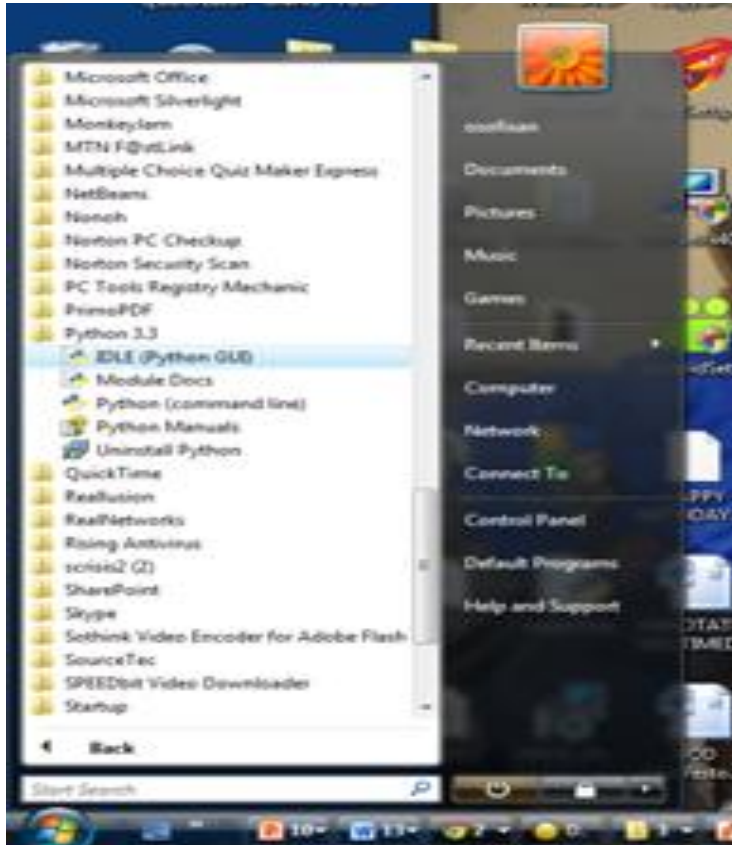
11.3 Features of Python Environment

When you install Python for your computer, you get a number of features:

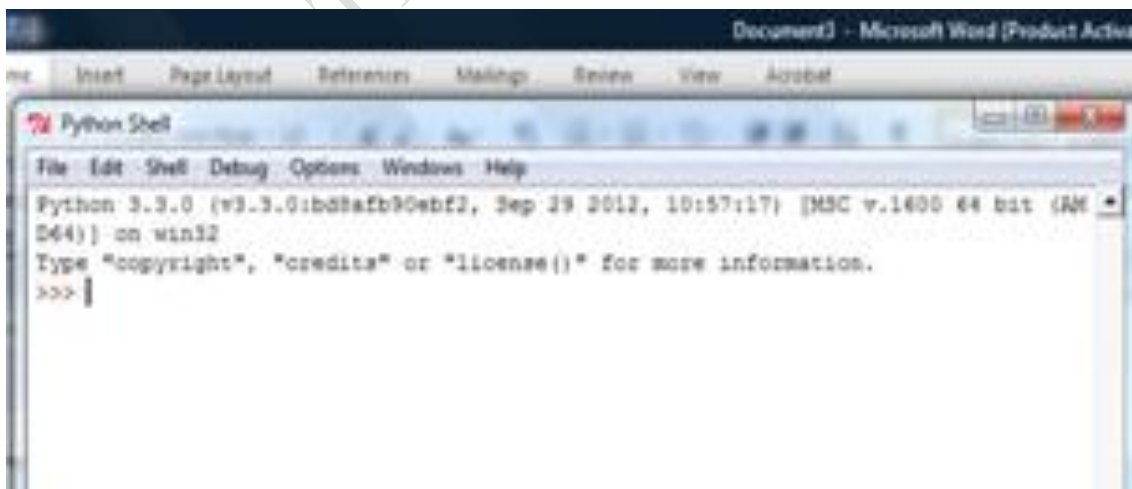
- (i) A Python shell, a window into which you can directly type Python commands and where interaction between you and the programs you write typically occurs.
- (ii) A simple editor called IDLE, in which you can type programs, update them, save them to disk and run them. IDLE’s interface is essentially the same on any machine you run it because it is a Python program!
- (iii) You get access to all the Python documentation on your local computer including:
 - A tutorial to get you started
 - A language reference for any details you might want to investigate
 - A library reference for modules you might wish to import
 - And use other nifty items

11.4 Starting the IDLE Python Interpreter

Let's get started with Python. To start the Python/IDLE combination on Windows, go to Start Menu>All Programs >Python 3.X > IDLE (Python GUI) as shown below:



This opens the Python Shell:



Of course, the example above is from Windows Vista. You can always locate Python 3.X from programs files in any version of Windows. If you are familiar with Python already and know other techniques for starting Python, you can still use those. Also, if you are using your own computer, you can also look into how to update PYTHON PATH, the list of folders/directories that Python searches for files

IDLE uses colored syntax to highlight your code. By default, built-in functions are purple, strings are green, and language keywords (like if) are orange. Any results produced are in blue. If you hate these color choices, don't worry; you can easily change them by adjusting IDLE's preferences.

IDLE also knows all about Python's indentation syntax, which requires code blocks be indented. When you start with Python, this can be hard to get used to, but IDLE keeps you straight by automatically indenting as needed.

IDLE is the Python's simple but powerful integrated development environment (IDE). It includes a color syntax-highlighting editor, a debugger, the Python Shell, and a complete copy of Python 3's online documentation set.

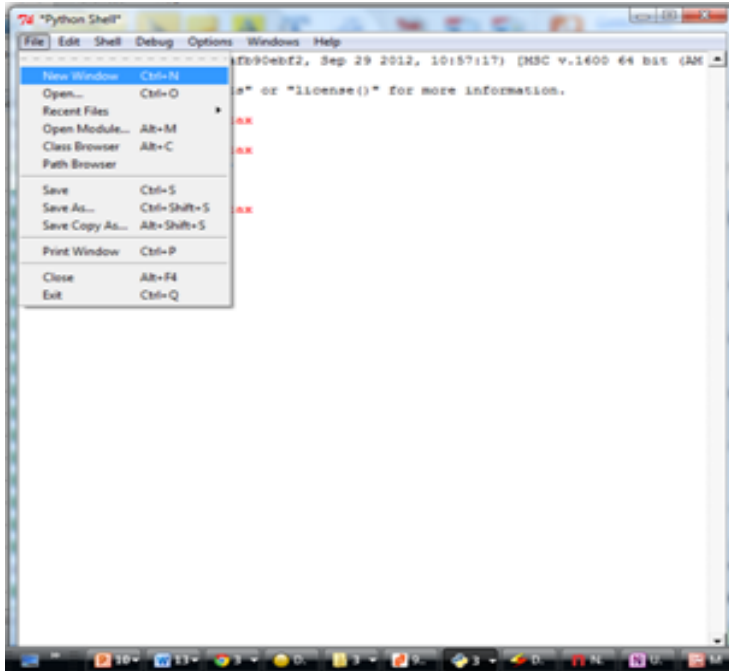
When you first start IDLE, you are presented with the "triple chevron" prompt (`>>>`) at which you enter code. The shell takes your code statement and immediately executes it for you, displaying any results produced on screen. IDLE knows all about Python syntax and offers "completion hints" that popup when you use a built-in function like `print()`. Python programmers generally refer to built-in functions as BIFs. The `print ()` BIF displays messages to standard output (usually the screen).

Class Exercise

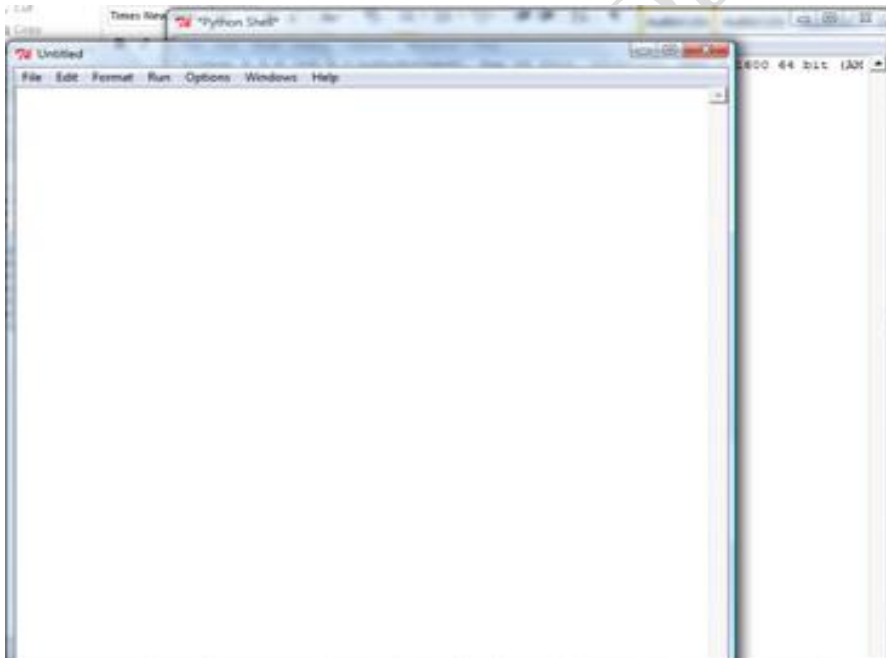
IDLE has lots of features, but you need to know about only a few of them to get going. Typing in some codes and then pressing the TAB key will cause IDLE to offer suggestions to help you complete your statement. Type `ex` and then press the TAB key. Write out the list of command completion suggestions that pop out.

11.5 Using Files

Typing into the shell is useful, but the commands that you write there are not saved as a file and therefore cannot be reused. We need to save our commands in a file so we can run the program again and again, and more importantly turn it in! Use the following steps to open a file as shown below. At the shell window, left-click on `File > New Window`



A second window will appear into which you can type python commands. This window is an editor window into which we can type our program



Follow the tradition again by typing the “Hello World” in the “Untitled” window and saving it as your first program

```
print("Hello World")
```

Save the program by left clicking in the Untitled Window the menu File >Save As

Save it as HelloWorld.py in Python3X folder of your computer

To run the program, select the editing window menu Run >Run Module. So what happens?

11.6 Python Variables

A variable is a label (a location in memory that points/holds one piece of data (one value). Value in them may change at any time. It is a location in memory that holds one piece of data. It is useful to think of a variable for now as a bucket holding one piece of data (a number or a string). The bucket has a unique name, and can only hold certain kinds of data. Consider the expression:

```
Price = 200
```

Price is a variable containing the value 200, and can contain only integers.

The following are the different types of variables are

(i) **Numbers:** (floating-point numbers and whole numbers). Examples are:

- -200 (integers- whole numbers without any decimal values)
- 20.90 (floating-point numbers)

(ii) **Text: (String).** This consists of one or more characters. Examples are:

- name = "Avatar"
- message = "Hello there"

11.7 Identifier

Identifiers are names or labels you can give to names of programs, variables or functions called identifier It is a series of characters consisting of letters, digits, and underscores (_). An identifier

- Must begin with a letter or underscore
- Does not begin with a digit
- Should not include spaces
- Letters are case sensitive in Python identifiers (Name is not the same as name)
- Examples: Welcome1, value, _value, button7
 - 2num is invalid
 - Butt on is invalid
- Python is case sensitive (capitalization matters)
 - a1 and A1 are different

11.8 Reserved Words/Keywords

Reserved words or *keywords* are words that have a specific meaning to the compiler and cannot be used in the program for any other purpose except for what they are meant for. The following are some of the reserved words in python:

and del from not while as elif global or with assert else if
pass yield break except import print class exec in raise continue finally is
return def for lambda try

11.9 Named Constants

Named constants are similar to variables: a memory location that has been given a name. Unlike variables, their contents *should not* change. The naming conventions for choosing variable names generally apply to constants but the name of constants should be all UPPER CASE. (You can separate multiple words with an underscore). They are capitalized so the reader of the program can distinguish them from variables. For some programming languages the translator will enforce the unchanging nature of the constant. For languages such as Python it is up to the programmer to recognize a constant for what it is and not to change it.

11.9.1 Named Constants versus Literals

A named constant is given an explicit name. For example, TAX_RATE is a named constant in these expressions:

- TAX_RATE = 0.2
- afterTax = income – (income * TAX_RATE)

A literal is an unnamed constant/magic number. It is not given a name, the value that you see is literally the value that you have. For example, 100000 and 0.2 are literals in the expression given below.

- afterTax = 100000 – (100000 * 0.2)

11.9.2 Advantages of Named Constants

1. They make your program easier to read and understand. Consider the expression:
populationChange = (0.1758 – 0.1257) * currentPopulation;

0.1758 and 0.1257 are magic numbers and numbers like this are meaningless. What do they represent? They should be avoided in programs whenever possible. Compare the expression with the codes below:

```
BIRTH_RATE = 17.58
MORTALITY_RATE = 0.1257
currentPopulation = 1000000
populationChange = (BIRTH_RATE - MORTALITY_RATE) * currentPopulation
```

2. Named constants make program easier to maintain. If the constant is referred to several times throughout the program, changing the value of the constant once will change it throughout the program. Examine the Python program below and try to confirm the importance of using named constants in it. One change in the initialization of the constants (BIRTH_RATE or MORTALITY_RATE) changes every reference to that constant.


```

BIRTH_RATE = 0.1758
MORTALITY_RATE = 0.1257
populationChange = 0
currentPopulation = 1000000
populationChange = (BIRTH_RATE - MORTALITY_RATE) * currentPopulation
if (populationChange > 0):
    print "Increase"
    print "Birth rate:", BIRTH_RATE, " Mortality rate:", MORTALITY_RATE, " Population
change:", populationChange
elif (populationChange < 0):
    print "Decrease"
    print "Birth rate:", BIRTH_RATE, " Mortality rate:", MORTALITY_RATE, "Population
change:", populationChange
else:
    print "No change"
    print "Birth rate:", BIRTH_RATE, " Mortality rate:", MORTALITY_RATE, "Population
change:", populationChange

```

Some programming languages have a mechanism for ensuring that named constants do not change. For example:

```

MY_CONSTANT = 100
MY_CONSTANT = 12

```

With programming languages that enforce the rule that constants can't change, this would result in an error. However, Python does not enforce the unchanging nature of constants so it is up to the person writing/modifying the program to avoid changing the value stored in a constant.

11.10 Displaying the Contents of Variables and Constants: Output

Format:

```

print (<variable name>)
print (<constant name>)

```

Example:

```

aNum = 10
A_CONSTANT = 10
print (aNum)
print (A_CONSTANT)

```

Output:

```
>>> ===== RESTART =====
>>>
10
10
>>> |
```

11.10.1 Displaying String Output

String output: A message appears onscreen that consists of a series of text characters. Whatever is contained with the quotes (single or double) is what appears onscreen.

Format:

```
print ("the message that you wish to appear")
```

OR

```
print ('the message that you wish to appear')
```

Example:

```
print ("Hello world")
```

```
print ('Welcome to the world of programming')
```

11.10.2 Mixed Output

Mixed output: This is getting string output and the contents of variables (or constants) to appear together.

Format:

```
print ("string", <variable or constant>, "string", <variable or constant> etc.)
```

Examples:

```
myInteger = 10
```

```
myReal = 10.5
```

```
myString = "hello"
```

```
print ("MyInteger:" , myInteger)
```

```
print ("MyReal:" , myReal)
```

```
print ("MyString:" , myString)
```

The comma signals to the translator that the string and the contents of the variable should appear on the same line

Output:

```
>>> ===== RESTART =====
>>>
MyInteger: 10
MyReal: 10.5
MyString: hello
>>> |
```

Summary

In this Session, you have been introduced to Python programming language at least in a little way. Some of the elements of programming as applied to Python were discussed.

Self Assessment Questions

1. What are the features of Python Programming Language?
2. Differentiate between variables and constants in programming
3. What is a named constant? Give two advantages of using named constants in programming.
4. Explain how Print statement works in Python Language

Study Session 12: Python Basic Operations

Expected Duration: 1 week or 2 contact hours

Introduction

In this Session, the various operations such as arithmetic and string that can be performed in Python are explained. Formatting of outputs are not left out.

Learning Outcomes

When you have studied this session, you should be able to explain:

- 12.1 The various Arithmetic Operations in Python
- 12.2 The various Numeric Data Types
- 12.3 How to input Data into the Computer
- 12.4 How to store Character Information
- 12.5 How to Convert Between Different Types of Information
- 12.6 How to Convert Between Different Types of Information: Getting Numeric Input
- 12.7 How to Determine the Type of Information Stored in a Variable
- 12.8 How to Format Your Outputs
- 12.9 Escape Codes
- 12.10 Program Documentation

12.1 Arithmetic Operations

The following is a table listing the different operators that can be used for arithmetic computations in Python Programming Language.

+	Addition $2 + 2 = 4$
*	Multiplication $2 * 2 = 4$
-	Subtractions $2 - 2 = 0$
/	Division $3 / 2 = 1.5$
**	Power $2 ** 3 = 8$
//	Division but will truncates decimal values (NO decimal values) $3 // 2 = 1$
%	MOD/ Modulo: Returns the remainder from a division $3 \% 2 = 1$

12.1.1 Order of Operation

When we have a mixture of these operators in an arithmetic expression, how will Python evaluate them?

()	Brackets (inner before outer)
**	Exponent
*, /, %	Multiplication, division, modulo
+, -	Addition, subtraction (BEMDAS)

- First level of precedence is top to bottom, i.e. Brackets first, followed by exponentiation
- Second level of precedence
 - If there are multiple operations that are on the same level then precedence goes from left to right. For example in `*, /, %`, `*` is evaluated first, next is `/` and then `%`
- Even for languages where there are clear rules of precedence (e.g., Java, Python) it is regarded as good style to explicitly bracket your operations: `x = (a * b) + (c / d)`

Simple program to add two numbers

```
# Program will initialize variables and add two numbers
# Specify values for num1 and num2 (initialization variables)
num1 = 2
num2 = 3
# Adding two numbers and storing result in a variable called "sum"
sum = num2 + num1
# Printing numbers and the sum of two numbers
print("num1 has a value of ", num1)
print("num2 has a value of ", num2)
print("The sum of num1 + num2 is ", sum)
```

```
>>> ===== RESTART =====
>>>
num1 has a value of 2
num2 has a value of 3
The sum of num1 + num2 is 5
>>>
```

12.2 Numeric Data Types

The information that is stored and manipulated by computers programs is referred to as *data*. There are two different kinds of numbers!

1. (5, 4, 3, 6) are whole numbers – they don't have a fractional part
2. (.25, .10, .05, .01) are decimal fractions

Inside the computer, whole numbers and decimal fractions are represented quite differently! We say that decimal fractions and whole numbers are two different *data types*.

The data type of an object determines what values it can have and what operations can be performed on it. Whole numbers are represented using the *integer* (*int* for short) data type. These values can be positive or negative whole numbers. Numbers that can have fractional parts are represented as *floating point* (or *float*) values.

How can we tell which is which?

1. A numeric literal without a decimal point produces an int value
2. A literal that has a decimal point is represented by a float (even if the fractional part is 0)

Python has a special function to tell us the data type of any value.

```
>>> type(3)
<class 'int'>
>>> type(3.1)
<class 'float'>
>>> type(3.0)
<class 'float'>
>>> myInt = 32
>>> type(myInt)
<class 'int'>
>>>
```

Operations on ints produce ints, operations on floats produce floats (except for /).

```
>>> 3.0+4.0
7.0
>>> 3+4
7
>>> 3.0*4.0
12.0
>>> 3*4
12
>>> 10.0/3.0
3.3333333333333335
>>> 10/3
3.3333333333333335
>>> 10 // 3
```

```
3
>>> 10.0 // 3.0
3.0
```

Integer division produces a whole number. That's why $10//3 = 3$! $10\%3 = 1$ is the remainder of the integer division of 10 by 3; $a = (a/b)(b) + (a\%b)$

12.3 Inputting Data into the Computer

Here we mean the computer program getting *string information* from the user. Strings cannot be used for calculations (information getting numeric input will be provided shortly).

Format:

```
<variable name> = input()
```

OR

```
<variable name> = input("<Prompting message>")
```

Example:

```
print ("What is your name: ")
```

```
name = input ()
```

OR

```
name = input ("What is your name: ")
```

On the computer all information is stored in binary (2 states)

Example: RAM memory stores information in a series of on-off combinations called bits
Information must be converted into binary to be stored on a computer.

12.4 Storing Character Information

Typically characters are encoded using ASCII. Each character is mapped to a numeric value

- E.g., 'A' = 65, 'B' = 66, 'a' = 97, '2' = 50

-

These numeric values are stored in the computer using binary

Character	ASCII numeric code	Binary code
'A'	65	01000001
'B'	66	01000010
'a'	97	01100001
'2'	50	00110010

Why is it important to know that different types of information are stored differently? It is because certain operations only apply to certain types of information and can produce errors or unexpected results when applied to other types of information.

Example:

```
num = input("Enter a number")
numHalved = num / 2
```

12.5 Converting Between Different Types of Information

Example motivation: you may want numerical information to be stored as a string (for the formatting capabilities) but also you want that same information in numerical form (in order to perform calculations). Some of the conversion mechanisms available in Python:

Format:

```
int (<value to convert>)
float (<value to convert>)
str (<value to convert>)
```

Examples:

```
x = 10.9
y = int(x) # Data Coercion or type casting
print(x, y)
```

```
>>> ----- RESTART -----
>>>
10.9 10
>>> |
```

Examples:

```
x = '100'
y = '-10.5'
print(x + y)
print(int(x) + float(y)) # PARSING
```

Output:

```
>>> ----- RESTART -----
>>>
100-10.5
89.5
```

Numeric to string:

```
aNum = 123
aString = str(aNum)
aNum = aNum + aNum
aString = aString + aString
print(aNum)
```



```
print (aString)
```

Output:

```
>>> ----- RESTART -----
>>>
246
123123
>>> |
```

12.6 Converting Between Different Types of Information: Getting Numeric Input

Because the 'input' function only returns string information it must be converted to the appropriate type as needed.

Example

```
# Problem!
HUMAN_CAT_AGE_RATIO = 7
age = input("What is your age in years: ")
catAge = age * HUMAN_CAT_AGE_RATIO
print ("Age in cat years: ", catAge)
```

Note:

- 'Age' refers to a string not a number.
- The '*' is not mathematical multiplication

```
# Problem solved!
HUMAN_CAT_AGE_RATIO = 7
age = int(input("What is your age in years: "))
catAge = age * HUMAN_CAT_AGE_RATIO
print ("Age in cat years: ", catAge)
```

Note:

- 'Age' converted to an integer.
- The '*' now multiplies a numeric value

A Comprehensive Example Program

```
#This program computes sum, average, standard deviation and mediation of 2 numbers
# Data Input
num1 = float(input("Enter the first number "))
num2 = float(input("Enter the Second number "))

# Data Processing
sum = num1 + num2
average = sum/2
```

```

#Printing results
print("Sum of the two numbers = " , sum)
print("Average of the two numbers = %.3f" %average)

#Computing the Standard Deviation
print("\n Computing the Standard Deviation...\n")
v = ((num1 - average)**2 + (num2 - average)**2)/2

from math import sqrt

sd = sqrt(v)
print("The standard deviation = %.2f" %sd)

# Computing the Mean Deviation
print("\n Computing the Mean Deviation...\n")
md = (abs(num1 - average) + abs(num2 - average))/2
print("The mean deviation = %.2f" %d)

stop = input("Press any key to continue....")

```

```

Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Enter the first number 3
Enter the Second number 5
Sum of the two numbers = 8.0
Average of the two numbers = 4.000

Computing the Standard Deviation....

The standard deviation = 1.00

Computing the Mean Deviation....

The mean deviation = 1.00
Press any key to continue....
>>>

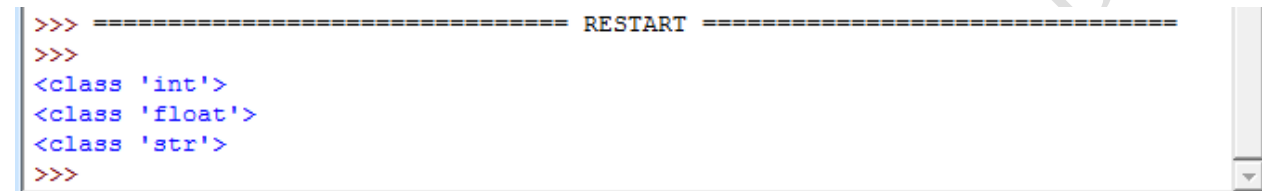
```

12.7 Determining the Type of Information Stored in a Variable

This can be done by using the pre-created python function ‘type’

Example program:

```
myInteger = 10
myString = "foo!"
print (type(myInteger))
print (type(10.5))
print (type(myString))
```



```
>>> ===== RESTART =====
>>>
<class 'int'>
<class 'float'>
<class 'str'>
>>>
```

12.8 Formatting Your Outputs

Output can be formatted in Python through the use of placeholders.

Format:

```
print (“%<type of info to display/code>” %<source of the info to display>)
```

Example:

```
num = 123
st = “GIS 733”
print (“num=%d” %num)
print (“course: %s” %st)
num = 12.5
print (“%f %d” %(num, num))
```

12.8.1 Types of Information That Can Be Displayed

Descriptor code	Type of Information to display
%s	String
%d	Integer (d = decimal / base 10)
%f	Floating point

12.8.2 Some Formatting Effects Using Descriptor Codes

Format:

`%<width>1.<precision>2<type of information>`

Examples:

```
num = 12.55
print ("%4.1f" % num)
print ("%0.1f" % num)
num = 12
st = "num="
print ("%s%d" % (st, num))
print ("%5s%5s%1s" % ("hi", "hihi", "there"))
```

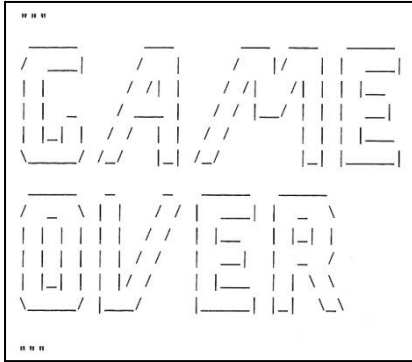
Note:

1. A positive integer will add leading spaces (right align), negatives will add trailing spaces (left align). Excluding a value will set the field width to a value large enough to display the output
2. For floating point representations only.

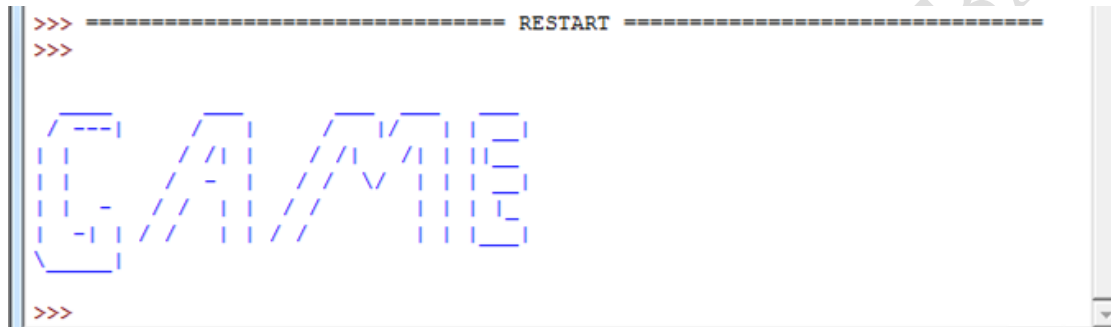
12.8.3 Triple Quoted Output

This is used to format text output. The way in which the text is typed into the program is exactly the way in which the text will appear onscreen.

```
# ASCII art that spells out "GAME OVER"
# Adapted from an example from "Python programming (second edition)
# by Michael Dawson
print ("""
  _____
 /---| / | / \| ||_
 ||   //| //| //||_
 ||   / - | // \| ||_
 || - // || // ||||_
 | -||// |//  ||||_
 \_____|
 """)
```



(From Python Programming (2nd Edition) by Michael Dawson)



12.9 Escape Codes

The back-slash character enclosed within quotes won't be displayed but instead indicates that a formatting (escape) code will follow the slash:

Escape sequence	Description
\a	Alarm. Causes the program to beep.
\b	Backspace. Moves the cursor back one space.
\n	Newline. Moves the cursor to beginning of the next line.
\t	Tab. Moves the cursor forward one tab stop.
\'	Single quote. Prints a single quote.
\"	Double quote. Prints a double quote.
\\	Backslash. Prints one backslash.

Run the code below and make your observations

```
print ("\a*Beep!*")
print ("h\bello")
print ("hi\nthere")
print ('it\s')
print ("he\y \"you\" ")
```

12.10 Program Documentation

Program documentation is used to provide information about a computer program to another *programmer* (writes or modifies the program). This is different from a user manual which is written for people who will *use the program*. Documentation is written inside the same file as the computer program (when you see the computer program you can see the documentation). The purpose is to help other programmers understand the program: what the different parts of the program do, what are some of its limitations etc. It doesn't contain instructions for the computer to execute. It doesn't get translated into machine language. It's information for the reader of the program:

- What does the program as a whole do e.g., tax program.
- What are the specific features of the program e.g., it calculates personal or small business tax.
- What are its limitations e.g., it only follows Canadian tax laws and cannot be used in the US. In Canada it doesn't calculate taxes for organizations with yearly gross earnings over \$1 billion.
 - What is the version of the program? If you don't use numbers for the different versions of your program then consider using dates (tie versions with program features).

Format:

```
# <Documentation>
```

The number sign '#' flags the translator that what's on this line is documentation.

Examples:

```
# Tax-It v1.0: This program will electronically calculate your tax return.
# This program will only allow you to complete a Canadian tax return
```

12.10.1 Types of Documentation

1. Header documentation

This is provided at the beginning of the program. It describes in a high-level fashion the features of the program as a whole (major features without a great deal of detail).

```
# HEADER DOCUMENTATION
# Word Processor features: print, save, spell check, insert images etc.

<program statement>
<program statement>
```

2. **Inline documentation**

This is provided throughout the program. It describes in greater detail the specific features of a part of the program.

```
# Documentation: Saving documents
# 'save': save document under the current name
# 'save as' rename the document to a new name
<program statement>
<program statement>

# Documentation: Spell checking
# The program can spell check documents using the following English variants:
# English (British), English (American), English (Canadian)
<program statement>
<program statement>
```

Summary

The various arithmetic and string operations in Python have been explained in this Session. Documentation of your programs is very important as it is very good for future program maintenance.

Self Assessment Questions

1. Write a program to compute your age given your year of birth and the current year.
2. Write a program to compute the area and volume of a solid cone. Look for the formulae that can be used to achieve your goal. Format your output to 2 decimal places.

Study Session 13: Control Structures in Python Language I: Making Decisions

Expected Duration: 1 week or 2 contact hours

Introduction

This Session is dedicated to Decision making in Python. The various control structures needed for decision making in programming are explained. In this Session, you will learn how to have your programs choose between alternative courses of action.

Learning Outcomes

When you have studied this session, you should be able to explain:

- 13.1 Why Branching/Decision Making is Needed?
- 13.2 Decision-Making in Python
- 13.3 Decision Making With an 'If-Else'
- 13.4 If Vs. If-Else
- 13.5 An Application Of Branches
- 13.6 Logical Operations
- 13.7 Nested Decision Making
- 13.8 Decision Making With Multiple If's
- 13.9 Decision Making With If-Elif-Else: Mutually Exclusive Conditions
- 13.10 Decision Making: Checking Matches

13.1 Why Is Branching/Decision Making Needed?

Decisions are questions with answers that are either true or false (Boolean) e.g., Is it true that the variable 'num' is positive? The program branches one way or another depending upon the answer to the question (the result of the Boolean expression).

Decision making in programming is very much needed for the following two reasons:

- 1) When alternative courses of action are possible and each action may produce a different result.
- 2) Branching/decision making can be used in a program to structure the alternatives and implement the results for each alternative.

13.2 Decision-Making In Python

Decision making/branching constructs (mechanisms) in Python are:

- **If**
- **If-else**
- **If-elif-else**

13.2.1 Decision Making With an 'If'

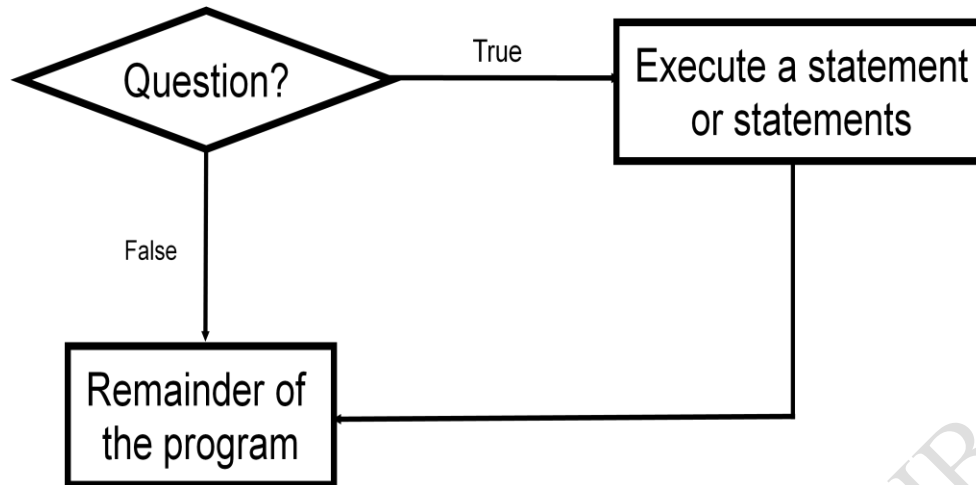


Figure 13.1 The 'If' Construct

The If construct is used when a question (Boolean expression) evaluates only to a true or false value (Boolean):

- If the question evaluates to true then the program reacts differently. It will execute a body after which it proceeds to the remainder of the program (which follows the if construct).
- If the question evaluates to false then the program doesn't react different. It just executes the remainder of the program (which follows the if construct).

Format:

(General format)

```
if (Boolean expression):  
    body
```

(Specific structure)

```
if (operand relational operator operand): # Boolean expression  
    body
```

Note: Indenting the body is mandatory!

Example:

```
if (age >= 18):  
    print ("You are an adult")
```

13.2.2 Allowable Operands for Boolean Expressions

Some operands that are allowed for Boolean expressions in Python are:

- integer
- floats (real)
- String

Make sure that you are comparing operands of the same type!

13.2.3 Allowable Relational Operators for Boolean Expressions

Python operator	Mathematical equivalent	Meaning	Example
<	<	Less than	5 < 3
>	>	Greater than	5 > 3
==	=	Equal to	5 == 3
<=	≤	Less than or equal to	5 <= 5
>=	≥	Greater than or equal to	5 >= 4
<>	≠	Not equal to	5 <> 5
OR			
!=			5 != 5

13.2.4 If (Simple Body)

The body of the *if* consists of a single statement

Format:

```
if (Boolean expression):  
    s1      # Body  
    s2
```

Indentation is used to indicate which group of statements forms the body of block of the *if* statement

Example:

```
if (num == 1):  
    print "Body of the if"  
    print "After body"
```

13.2.5 If (Compound Body)

Body of the *if* consists of multiple statements

Format:

```
if (Boolean expression):  
    s1  
    s2  
    :  
    sn  
sn+1
```

End of the indentation denotes the end of decision-making

Example:

```
taxCredit = 0
taxRate = 0.2
income = float(input("What is your annual income: "))
if (income < 10000):
    print ("Eligible for social assistance")
    taxCredit = 100
tax = (income * taxRate) - taxCredit
print ("Tax owed N%.2f" %tax)
```

13.3 Decision Making With an 'If-Else'

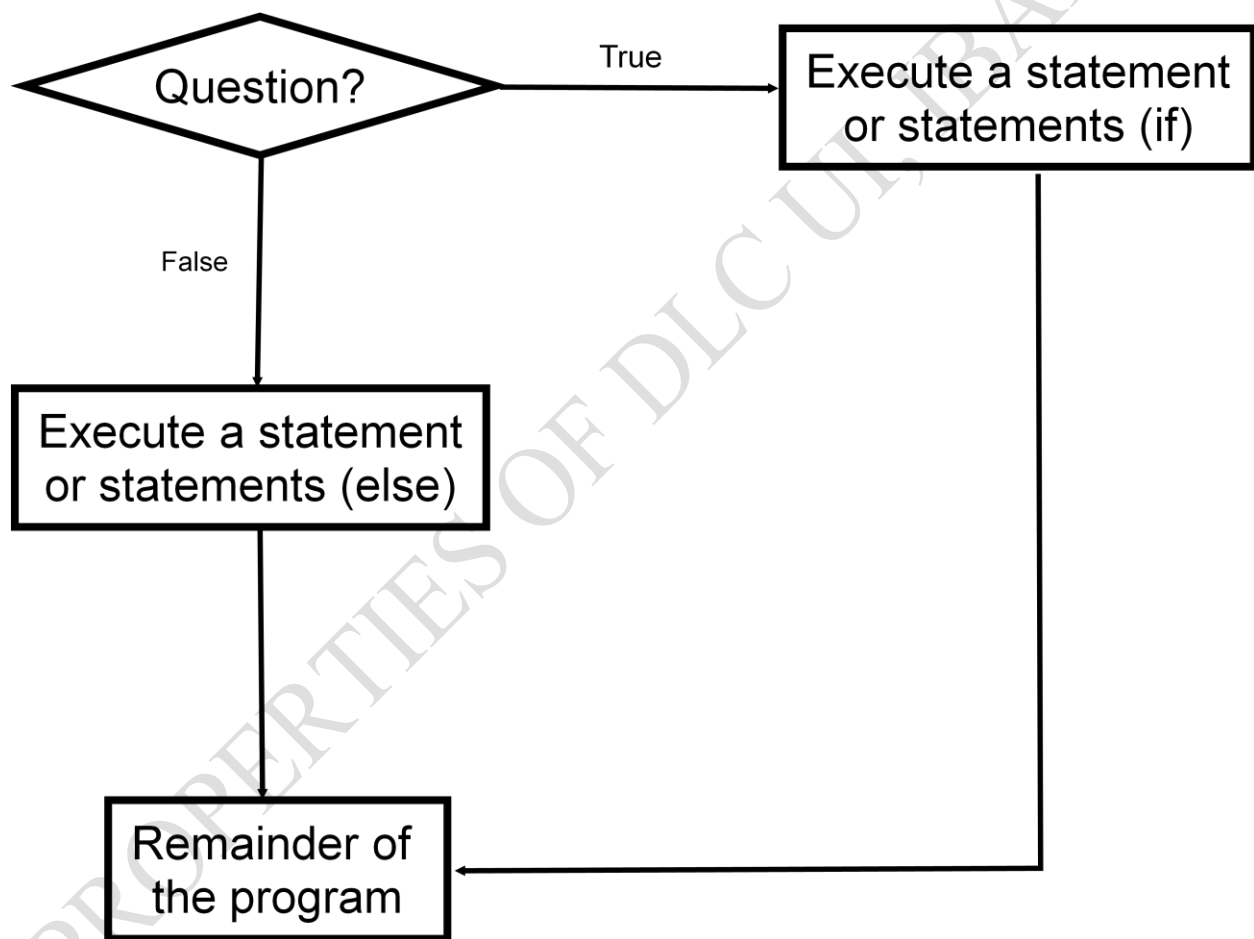


Figure 13.2 The If-Else Construct

In this type of decision making, the program checks if a condition is true (in which case something should be done) but also reacts if the condition is not true (false).

Format:

```
if (operand relational operator operand):  
    body of 'if'  
else:  
    body of 'else'  
    additional statements
```

Example:

```
if (age < 18):  
    print ("Not an adult")  
else:  
    print ("Adult")  
print ("Tell me more about yourself")
```

Example 2:

```
if (income < 10000):  
    print ("Eligible for social assistance")  
    taxCredit = 100  
    taxRate = 0.1  
else:  
    print ("Not eligible for social assistance")  
    taxRate = 0.2  
tax = (income * taxRate) - taxCredit
```

Example 3: Solving Quadratic Equation

```
from math import sqrt # Including square root function from the Math Library  
print("This program computes the roots of any quadratic equation\n")  
# The input data  
a = float(input("Enter the value of the coefficient of x-square, a "))  
b = float(input("Enter the value of the coefficient of x, b "))  
c = float(input("Enter the constant, c "))  
  
#Computing the discriminant, d  
d = b**2 - 4 * a * c  
  
#Determining the roots  
if d < 0:  
    print("Complex roots please.....")  
elif d == 0:  
    print("Equal roots with values....")  
    r = -b/(2 *a)  
    print("%.1f" %r)  
else:  
    print("Two real roots with values ....")
```

```

r1 = (-b + sqrt(d))/(2 * a)
r2 = (-b - sqrt(d))/(2 * a)
print("%.1f and %.1f" %(r1, r2))
print(" Thanks for using the program")
print("Press any key to continue....")

```

Sample output

```

>>>
This program computes the roots of any quadratic equation

Enter the value of the coefficient of x-square, a 2
Enter the value of the coefficient of x, b -3
Enter the constant, c 1
Two real roots with values ....
1.0 and 0.5
Thanks for using the program
Press any key to continue....
>>>

```

Class Exercise 1

A shop sells only one product at a unit price P. If a customer buys at least 5 quantities, s/he is given a 5% discount. Implement a Python program to compute

- (i) The gross pay before discount
- (ii) The Discount given
- (iii) The net pay after discount

13.4 If Vs. If-Else

The *if*:

- (a) Evaluates a Boolean expression (ask a question).
- (b) If the expression evaluates to true then it executes the 'body' of the if.
- (c) No additional action is taken when the expression evaluates to false.
- (d) Is used when your program is supposed to react differently only when the answer to a question is true (and do nothing different if it's false).

The *If-Else*:

- (a) Evaluates a Boolean expression (ask a question)
- (b) If the expression evaluates to true then it executes the 'body' of the if.
- (c) If the expression evaluates to false then it executes the 'body' of the else.
- (d) Is used when your program is supposed to react differently for both the true and the false cases.

Class Exercise 2

CLASS Nig. Ltd has just employed you to computerize her Salaries Section. Your analysis of the company shows that every staff is given a monthly basic salary commensurate with his/her job description. In addition, the following allowances are given to the staff:

- 10% of Basic Salary as Housing Allowance.
- 7½ % of Basic Salary as Transport Allowance.
- If the grade level of a staff is above GL/07, the staff is entitled to Entertainment Allowance of 2% of his/her Basic Salary.
- A staff that has spent up to 10 years in the company is also given a 1.5% of his/her Basic Salary as “long Serving Allowance”.
- Every staff is entitled to N500 Meal Subsidy Allowance per month.

Write a Pay Roll program for the Company, using Python programming language.

13.5 An Application of Branches

Branching statements can be used to check the validity of data (if the data is correct or if it's a value that's allowed by the program).

General structure:

```
if (error condition has occurred)
    React to the error
```

Example:

```
if (age < 0):
    print "Age cannot be a negative value"
```

13.6 Logical Operations

There are many logical operations but the three most commonly in computer programs include:

- (a) Logical AND
- (b) Logical OR
- (c) Logical NOT

13.6.1 Logical AND

The popular usage of the AND applies when *ALL* conditions must be met.

Example: Pick up your son (condition1) AND pick up your daughter (condition2) after school today.

Logical AND can be specified more formally in the form of true table.

Truth table (AND)		
C1	C2	C1 AND C2
False	False	False
False	True	False
True	False	False
<i>True</i>	<i>True</i>	<i>True</i>

13.6.2 Logical OR

The correct everyday usage of the OR applies when *AT LEAST* one condition must be met.

Example: You are using additional recommended resources for this course: the online textbook (Condition1) OR the paper textbook (Condition 2) available in the bookstore.

Similar to AND, logical OR can be specified more formally in the form of true table.

Truth table (AND)		
C1	C2	C1 OR C2
False	False	False
<i>False</i>	<i>True</i>	<i>True</i>
<i>True</i>	<i>False</i>	<i>True</i>
<i>True</i>	<i>True</i>	<i>True</i>

13.6.3 Logical NOT

The everyday usage of logical NOT negates (or reverses) a statement.

Example: I am finding this class quite stimulating and exciting (statement).....*NOT!!! (Negation)*

The truth table for logical NOT is quite simple:

Truth table	
S	Not S
False	True
True	False

Note that logic operators can be used in conjunction with branching. Typically the logical operators AND, OR are used with multiple conditions:

- If multiple conditions *must all be met* before a statement will execute. (AND)
- If *at least one condition* must be met before a statement will execute. (OR)

The logical NOT operator can be used to check for inequality (not equal to).

E.g., If it's true that the user *did not* enter an invalid value the program can proceed.

13.6.4 Decision-Making with Multiple Expressions

Format:

if (Boolean expression) logical operator (Boolean expression):
body

Example 1: with and

```
if (x > 0) and (y > 0):
    print ("X is positive, Y is positive")
```

```
if (yearsOnJob <= 2) and (salary > 50000):
    print ("You are fired")
```

Example 2: with or

```
if (gpa > 3.7) or (yearsJobExperience > 5):
    print ("You are hired")
```

13.6.4.1 Quick Summary: Using Multiple Expressions

Use multiple expressions when multiple questions must be asked and the result of each expression may have an effect on the other expressions:

AND:

- All Boolean expressions must evaluate to true before the entire expression is true.
- If any expression is false then whole expression evaluates to false

OR:

- If any Boolean expression evaluates to true then the entire expression evaluates to true.
- All Boolean expressions must evaluate to false before the entire expression is false.

13.7 Nested Decision Making

What happens when decision paths are nested within one another? In this case, Decision making is dependent on one another. The first decision must evaluate to true before successive decisions are even considered for evaluation.

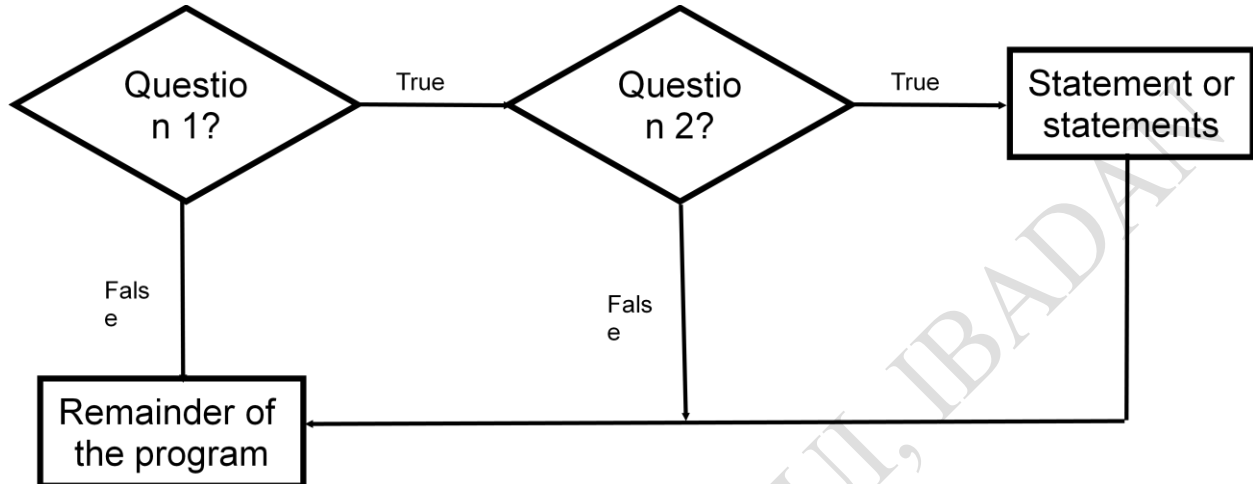


Figure 13.3: Nested Decision Making

One decision is made inside another. Outer decisions must evaluate to true before inner decisions are even considered for evaluation.

Format:

```
if (Boolean expression): # Outer body starts
    if (Boolean expression):
        inner body
```

Example:

```
if (income < 10000):
    if (citizen == 'y'):
        print ("This person can receive social assistance")
        taxCredit = 100
tax = (income * TAX_RATE) - taxCredit
```

Class Discussion

What's the difference between employing nested decision making and a logical AND?

13.8 Decision Making With Multiple If's

If there are multiple *ifs* in a program, any, all or none of the conditions may be true (independent)

Format:

```
if (Boolean expression 1):
    body 1
if (Boolean expression 2):
```

body 2
:
statements after the conditions

Example:

```
if (num1 > 0):  
    print "num1 is positive"  
if (num2 > 0):  
    print "num2 is positive"  
if (num3 > 0):  
    print "num3 is positive"
```

Multiple If's indicates mutually exclusive conditions

- At most *only one* of many conditions can be true
- Can be implemented through multiple if's

Example:

```
if (gpa == 4):  
    letter = 'A'  
if (gpa == 3):  
    letter = 'B'  
if (gpa == 2):  
    letter = 'C'  
if (gpa == 1):  
    letter = 'D'  
if (gpa == 0):  
    letter = 'F'
```

13.9 Decision Making With If-Elif-Else: Mutually Exclusive Conditions

In this control, several conditional paths exist for testing but only one of them is feasible at a time.

Format:

```
if (Boolean expression 1):  
    body 1  
elif (Boolean expression 2):  
    body 2  
:  
else  
    body n  
statements after the conditions
```

Example:

```

if (gpa == 4):
    letter = 'A'
elif (gpa == 3):
    letter = 'B'
elif (gpa == 2):
    letter = 'C';
elif (gpa == 1):
    letter = 'D'
elif (gpa == 0):
    letter = 'F'
else:
    print ("GPA must be one of '4', '3', '2', '1' or '1'")

```

Note that the body of the else executes only when all the Boolean expressions are false. (Useful for error checking/handling).

Recap: What Decision Making Constructs Are Available In Python/When To Use Them

Construct	When To Use
If	Evaluate a Boolean expression and execute some code (body) if it's true
If-else	Evaluate a Boolean expression and execute some code (first body 'if') if it's true, execute alternate code (second body 'else') if it's false
Multiple if's	Multiple Boolean expressions need to be evaluated with the answer for each expression being independent of the answers for the others (non-exclusive). Separate code (bodies) can be executed for each expression.
If-elif-else	Multiple Boolean expressions need to be evaluated but zero or at most only one of them can be true (mutually exclusive). Zero bodies or exactly one body will execute. Also it allows for a separate body (else) to execute when all the if-elif Boolean expressions are false.
Compound decision making	More than one Boolean expression must be evaluated before some code (body) can execute. All expressions must evaluate to true (AND) or at least one expression must evaluate to true (OR).
Nested decision making	The outer Boolean expression must be true before the inner expression will even be evaluated. (Inner Boolean expression is part of the body of the outer Boolean expression).

13.10 Decision Making: Checking Matches

Python provides a quick way of checking for matches within a set. E.g., for a menu driven program is the user's response one of the values in the set of valid responses.

Format:

(Strings)

```
if <string variable> in "<string1> <string2>...<stringn>":  
    body
```

(Numeric)

```
if <numeric variable> in (<number1>, <number2>,...<numbern>"):  
    body
```

Example:

(String):

```
if answer in "password1 password2 password3":  
    print "correct"  
else:  
    print "incorrect"
```

(Numeric):

```
if num in (1, 2, 3):  
    print "in set"
```

Summary

At the end of this Session, you should now know

(i) The decision making constructs available in Python:

- If
- If-else
- If-elif-else

How each one works and when should each one be used

(ii) Three logical operations:

- AND
- OR
- NOT

(iii) How to evaluate and use decision making constructs:

By tracing the execution of simple decision making constructs

(iv) How to evaluate nested and compound decision making constructs and when to use them

(v) How the bodies of the decision making construct are defined:

- What is the body of decision making construct
- What is the difference between decision making constructs with simple bodies and those with compound bodies

(vi) How multiple expressions are evaluated and how the different logical operators work

Self Assessment Question

A shop sells only one product at a unit price P . If a customer buys between 1 and 5 quantities, s/he is given a 2% discount. If s/he buys between 6 to 10, 4% discount is given. Finally, if s/he buys more than 10 quantities, 10% discount is given. Implement a program to compute

- (i) The gross pay before discount
- (ii) The Discount given
- (iii) The net pay after discount

PROPERTIES OF DLC UI, IBADAN

Study Session 14: Control Structures in Python Language II: Loops in Python

Expected Duration: 1 week or 2 contact hours

Introduction

In this Session, you will learn how to rerun parts of your program without having to duplicate the code.

Learning Outcomes

When you have studied this session, you should be able to explain:

- 14.1 Basic Structure of Loops
- 14.2 Types of Loops
- 14.3 Pre-Test Loops in Python
- 14.4 Infinite Loops
- 14.5 Sentinel Controlled Loops
- 14.6 What Looping Constructs Are Available In Python/When To Use Them
- 14.7 Nested Loops

14.1 Basic Structure of Loops

Whether or not a part of a program repeats is determined by a loop control (typically just a variable). The structure of a loop looks like the following

- Initialize the control to the starting value
- Test the control against a stopping condition (Boolean expression)
- Execute the body of the loop (the part to be repeated)
- Update the value of the control

14.2 Types of Loops

There are two types of loops in programming:

1. Pre-test loops

In this type of loop,

1. Initialize loop control
2. Check if the stopping condition has been met
 - a. If it's been met then the loop ends
 - b. If it hasn't been met then proceed to the next step
3. Execute the body of the loop (the part to be repeated)
4. Update the loop control
5. Go to step 2

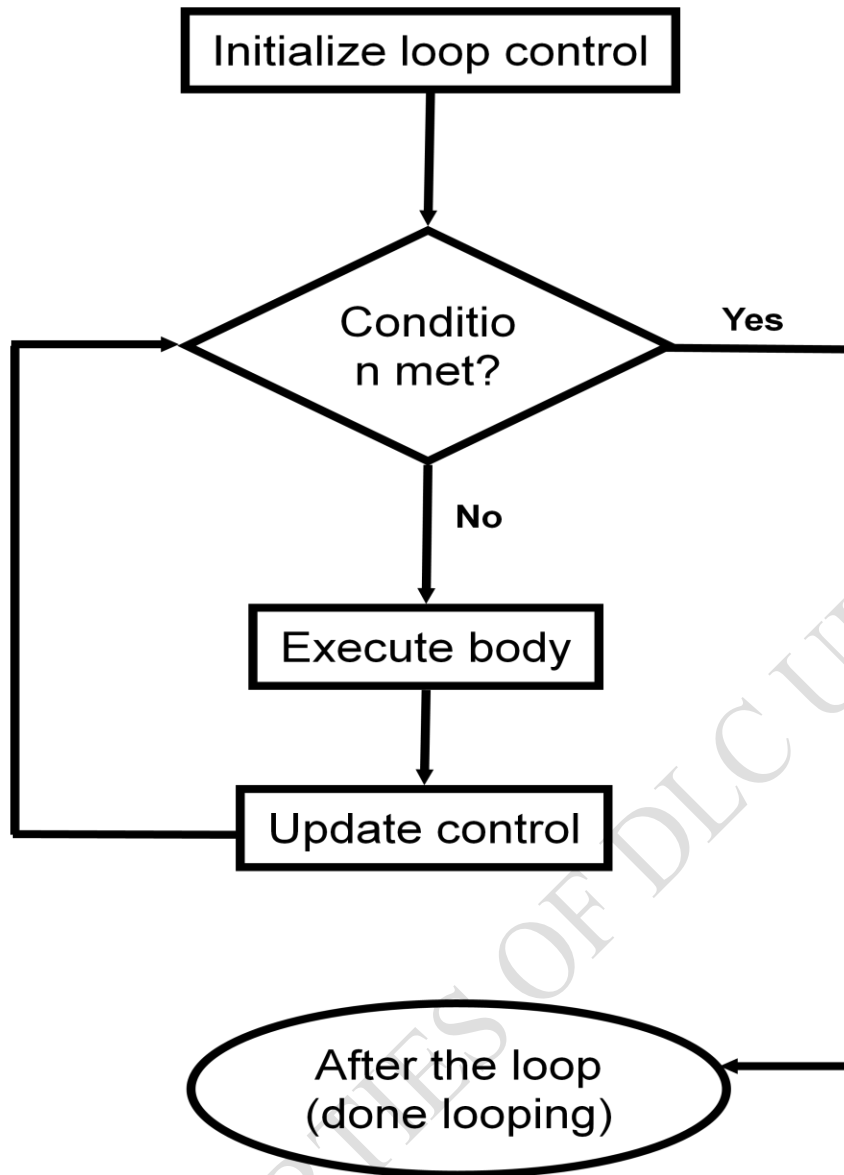


Figure 14.1: Pre-test Loops

2. Post-test loops

In this type of loop,

- We check the stopping condition *after* executing the body of the loop.
- The loop executes *one or more* times.

In post-test loops,

1. Initialize loop control (sometimes not needed because initialization occurs when the control is updated)
2. Execute the body of the loop (the part to be repeated)
3. Update the loop control
4. Check if the stopping condition has been met
 - a. If it's been met then the loop ends

- b. If it hasn't been met then return to step 2.

In practice, this type of loop is not implemented in Python.

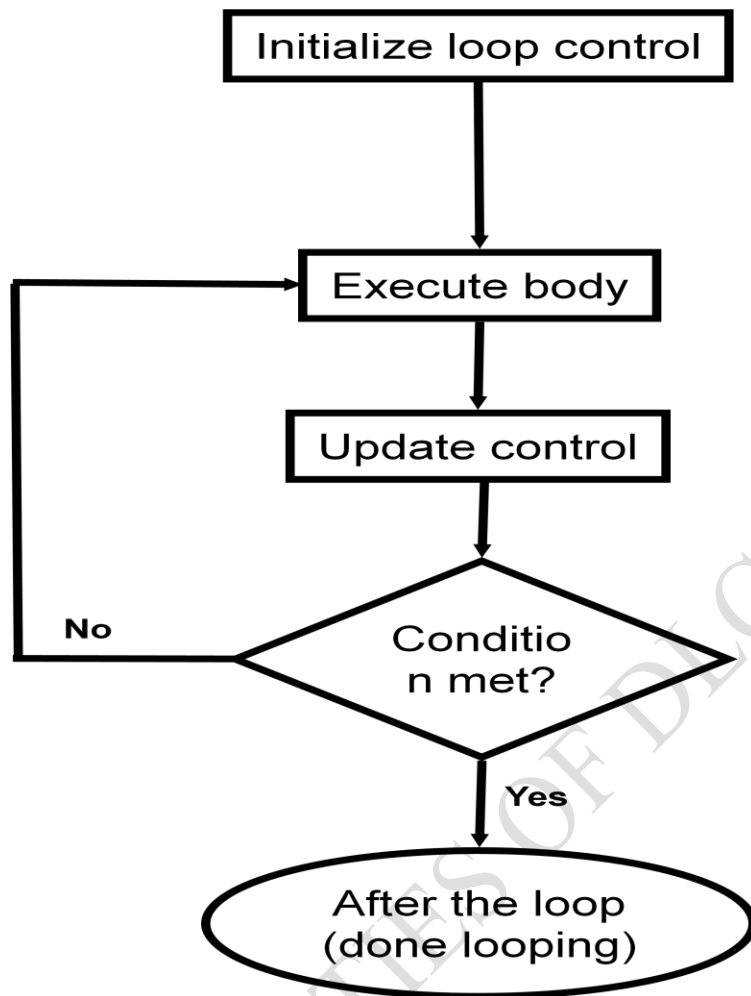


Figure 14.2 Post-test loops

14.3 Pre-Test Loops in Python

There are only two pre-test loops implemented in Python. They are:

1. While
2. For

The general characteristics of these loops are:

1. The stopping condition is checked *before* the body executes.
2. These types of loops execute zero or more times.

14.3.1 The While Loop

This type of loop can be used if it is *not known* in advance how many times that the loop will repeat. It is the most powerful type of loop; any other type of loop can be simulated with a while loop

Format:

Simple conditional while loop:

```
while (Boolean expression):  
    body
```

Compound conditional while loop:

```
while (Boolean expression) Boolean operator (Boolean expression):  
    body
```

Note the use of colon (:) at the end of the *while loop* header and the indentation of statements under the *while loop* block

Example 1: Run the programs below and test with values of I = 2, 7, 60 and -2. Record your observations

```
i = 1  
while (i <= 4):  
    print ("i =", i)  
    i += 1    # or i = i + 1  
print ("Done!")
```

```
I = 2  
sum = 0  
while (I <= 50):  
    print("I = ", I)  
    sum += I  
    I += 2  
print("Total sum = ", sum )
```

Example program

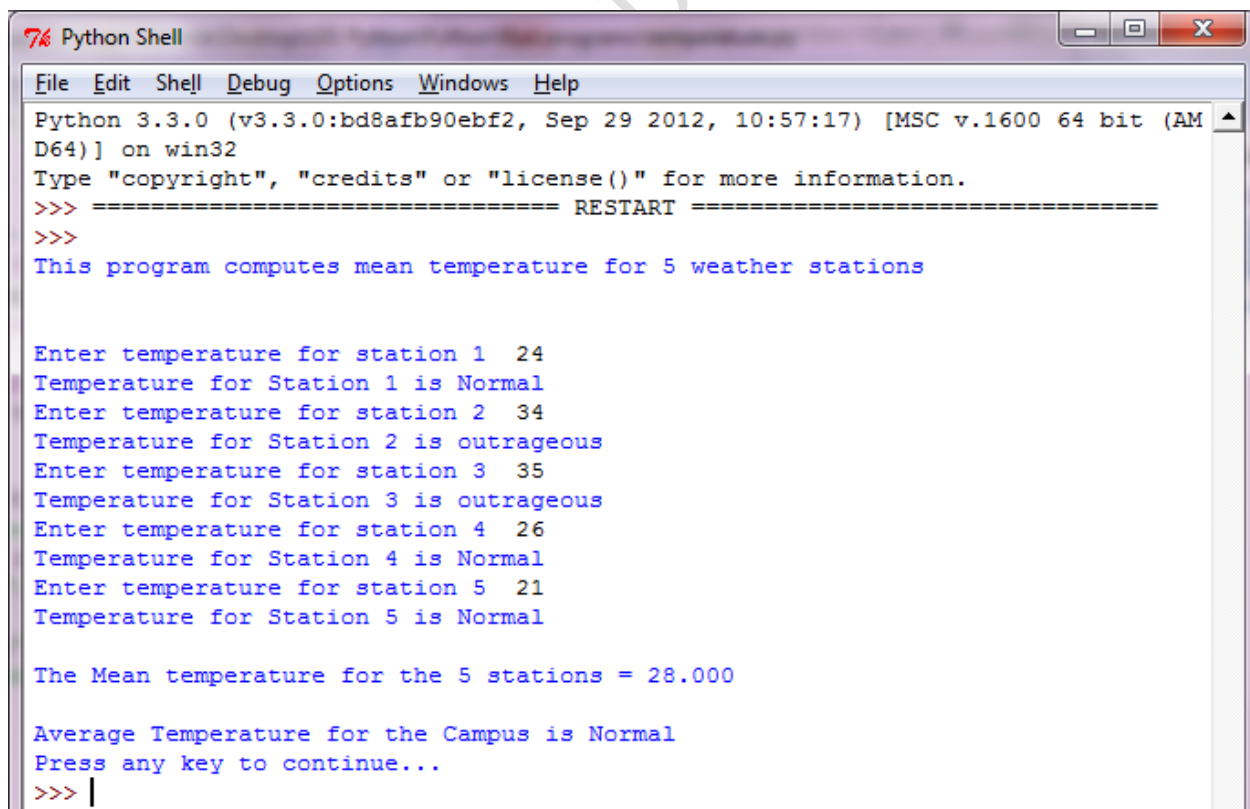
1. There are 5 weather stations on campus. Write a program using while loop to report if temperature at a station is outrageous (temp > 28) or normal.
2. Extend the program to compute the average temperature for all the stations

#Mean Temperature for 5 Weather Stations

```
print("This program computes mean temperature for 10 weather stations\n")
print( )
i = 1
sum = 0
while (i <= 5):
    temp = float(input("Enter temperature for station %d " %i))
    if (temp > 28):
        print("Temperature for Station %d is outrageous" %i)
    else:
        print ("Temperature for Station %d is Normal" %i)
    sum += temp
    i += 1 # End of While loop block

mean = sum/5

print("\nThe Mean temperature for the 5 stations = %.3f " %mean)
if (mean > 28):
    print("\nAverage Temperature for the campus is outrageous" )
else:
    print ("\nAverage Temperature for the Campus is Normal" )
print("Press any key to continue...")
```



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
This program computes mean temperature for 5 weather stations

Enter temperature for station 1 24
Temperature for Station 1 is Normal
Enter temperature for station 2 34
Temperature for Station 2 is outrageous
Enter temperature for station 3 35
Temperature for Station 3 is outrageous
Enter temperature for station 4 26
Temperature for Station 4 is Normal
Enter temperature for station 5 21
Temperature for Station 5 is Normal

The Mean temperature for the 5 stations = 28.000

Average Temperature for the Campus is Normal
Press any key to continue...
>>> |
```

Class Exercises

Adjust the above program to compute the average of all outrageous temperatures for the campus

14.3.2 The For Loop

The *for* loop is typically used when it *is known* in advance how many times that the loop will execute (counting loop).

14.3.2.1 Up-Counting *for* Loop

Syntax:

```
for <name of loop control> in <something that can be iterated>:  
    body
```

Every for loop usually has three parts: the initial value to start the counting, the final stopping criterion to end the loop and the increment. Python behaves in a bizarre way in the counting. Consider the following code whose goal is to print numbers from 1 to a point specified by the user.

```
n = int(input("What is the stopping number? "))  
total = 0  
for i in range (1,n,1):  
    print ("i=", i)  
    print ("Done!")
```

Sample output:

```
>>>  
What is the stopping number? 5  
i= 1  
i= 2  
i= 3  
i= 4  
Done!  
>>>
```

What did you notice in the output? The counting did not reach 5 in the iterations. So, what do we do? We just need to add 1 to the n in order for the loop to reach 5 iterations. By the way, if the step of the counting is going to be 1, we simply ignore the third part, the step.

```
n = int(input("What is the stopping number? "))  
total = 0  
for i in range (1,n +1):  
    print ("i=", i)  
    print ("Done!")
```

Sample Output

```
>>>
What is the stopping number? 5
i= 1
i= 2
i= 3
i= 4
i= 5
Done!
>>>
```

14.3.2.2 Counting Down With A For Loop

```
print("\n Downward Counting loop to 1 \n\n")
n = int(input("What is the starting number? "))
total = 0
for i in range (n, 0, -1):
    print ("i = ", i)
print ("Done!")
```

Sample Output:

```
>>>
Downward Counting loop to 1
What is the starting number? 5
i = 5
i = 4
i = 3
i = 2
i = 1
Done!
>>>
```

Note in the above program, we intend to stop the counting at 1, but we have to subtract 1 from the stopping criterion. This is again a peculiarity of Python programming language.

Check the two programs below

Program 1:

```
print("Downward Counting loop to a number \n\n")
n = int(input("What is the starting number? "))
m = int(input("What is the stopping number? "))
total = 0
for i in range (n, m, -1):
    print ("i = ", i)
print ("Done!")
```

Sample Output

```
>>>
Downward Counting loop to a number
What is the starting number? 10
What is the stopping number? 6
i = 10
i = 9
i = 8
i = 7
Done!
>>>
```

Program 2:

```
print("Downward Counting loop to a number \n\n")
n = int(input("What is the starting number? "))
m = int(input("What is the stopping number? "))
total = 0
for i in range (n, m-1, -1):
    print ("i = ", i)
print ("Done!")
```

Sample Output:

```
>>>
Downward Counting loop to a number
What is the starting number? 10
What is the stopping number? 6
i = 10
i = 9
i = 8
i = 7
i = 6
Done!
>>>
```

Exercise:

Implement a program to compute n-factorial n!
 $N! = 2 \times 3 \times 4 \times \dots \times N$

A sample solution is shown here:

```
n = int(input("What is the value of n? "))
f = 1
for i in range (1, n+1, 1):
    f = f * i
print ("Factorial of %d = %d " %(n,f))

print("\n It worked using up counting loop and adding 1 to n")
```

```

print("\n Now Using Down Counting loop....")
n = int(input("What is the value of n? "))
f = 1
for i in range (n, 1, -1):
    f = f * i
print ("Factorial of %d = %d " %(n,f))

```

Sample output:

```

>>>
What is the value of n? 5
Factorial of 5 = 120

```

It worked using up counting loop and adding 1 to n

```

Now Using Down Counting loop....
What is the value of n? 5
Factorial of 5 = 120
>>>

```

Now adapt the program to compute combination of N and R
 $NCR = N! / ((N - R)! * R!)$

Note: Loop Increments Need Not Be Limited To One

Exercise: Write a program to compute sum of multiples of 5 from 1 to m, where m is a valid integer.

14.4 Infinite Loops

infinite loops never end (the stopping condition is never met). They can be caused by logical errors in following ways:

- The loop control is never updated
- The updating of the loop control never brings it closer to the stopping condition

Example:

```

i = 1
while (i <= 10):
    print ("i = ", i)
    i = i + 1

```

Question: Why would the above program not terminating when executed?

14.5 Sentinel Controlled Loops

The stopping condition for the loop occurs when the ‘sentinel’ value is reached. A sentinel is a value that terminates a loop.

Example 1:

```

total = 0
temp = 0
while (temp >= 0):
    temp = input ("Enter a non-negative integer (negative to end series):")
    temp = int(temp)
    if (temp >= 0):
        total = total + temp
print ("Sum total of the series:", total)

```

Sentinel controlled loops are frequently used in conjunction with the error checking of input.

Example 2:

```

selection = " "
while selection not in ("a", "A", "r", "R", "m", "M", "q", "Q"):
    print "Menu options"
    print "(A)dd a new player to the game"
    print "(R)emove a player from the game"
    print "(M)odify player"
    print "(Q)uit game"
    selection = raw_input ("Enter your selection: ")
if selection not in ("a", "A", "r", "R", "m", "M", "q", "Q"):
    print "Please enter one of 'a', 'r', 'm' or 'q'"

```

14.6 Recap: What Looping Constructs Are Available In Python/When To Use Them

Construct	When To Use
Pre-test loops	You want the stopping condition to be checked before the loop body is executed (typically used when you want a loop to execute zero or more times).
While	The most powerful looping construct: you can write a 'while-do' loop to mimic the behavior of any other type of loop. In general it should be used when you want a pre-test loop which can be used for most any arbitrary stopping condition e.g., execute the loop as long as the user doesn't enter a negative number.
For	A 'counting loop': You want a simple loop to repeat a certain number of times.
Post-test: None in Python	You want to execute the body of the loop before checking the stopping condition (typically used to ensure that the body of the loop will execute at least once). The logic can be simulated in Python however.

14.7 Nested Loops

In these types of loops, one loop is executed inside of another loop(s). The inner loop runs faster than the outer loop. In the example structure shown below, for every single value of n of the outer loop, inner loop runs m times

Example structure:

Outer loop (runs n times)

 Inner loop (runs m times)

 Body of inner loop (runs $n \times m$ times)

Example Program:

```
for i in range (1, 3, 1):
    for j in range (1, 4, 1):
        print ("i = ", i, " j = ", j)
print ("Done!")
```

Sample Output:

```
>>>
i = 1 j = 1
i = 1 j = 2
i = 1 j = 3
i = 2 j = 1
i = 2 j = 2
i = 2 j = 3
Done!
```

Summary

No doubt you have been introduced to the available loop structures in Python, such as *for* and *while* pre-test loops. Post-test loops such as *do – while* are not implemented in Python for now. The bizarre way in which Python handles counting in *for* loops is also demonstrated. Now answer the following questions.

Self-Assessment Questions

1. Write a program to compute the sum of all numbers greater than 5 in a given set of n numbers, where n is a valid integer.
2. In an experiment in a Physics Laboratory, there are n experimental set-ups, each being repeated m times. Write a program to compute the mean values obtained from each experimental set-up.

Study Session 15: Functions in Python Language

Expected Duration: 1 week or 2 contact hours

Introduction

Breaking down a problem into modules enhances program readability and maintenance. These modules are called functions in programming. A function solves a program module and returns a single result or output to the main program. In this Session, you shall be introduced to the writing and inclusion of functions in your programs.

Learning Outcomes

When you have studied this session, you should be able to explain:

- 15.1 Functions: Decomposition and Code Reuse
- 15.2 Procedural Programming
- 15.3 Decomposing A Problem Into Procedures
- 15.4 Things Needed In Order to Use Functions
- 15.5 Common Mistakes with Functions in Python
- 15.6 Where to Create Local Variables
- 15.7 Default Parameters
- 15.8 Good Style: Functions
- 15.9 Parameter Passing
- 15.10 Scope of a Variable
- 15.11 Global Scope
- 15.12 Function Pre-Conditions
- 15.13 Function Post-Conditions
- 15.14 Why Employ Problem Decomposition and Modular Design

15.1 Functions: Decomposition and Code Reuse

Large problems are decomposed into modules or functions to reduce program size by creating reusable sections or units. There are two methods of designing programs: Top down and bottom up approaches.

15.1.1 Top Down Design

In top down design approach,

1. We start by outlining the major parts (structure) of the problem. Consider the task of writing your autobiography. The task can be broken down as in Figure 15.1.

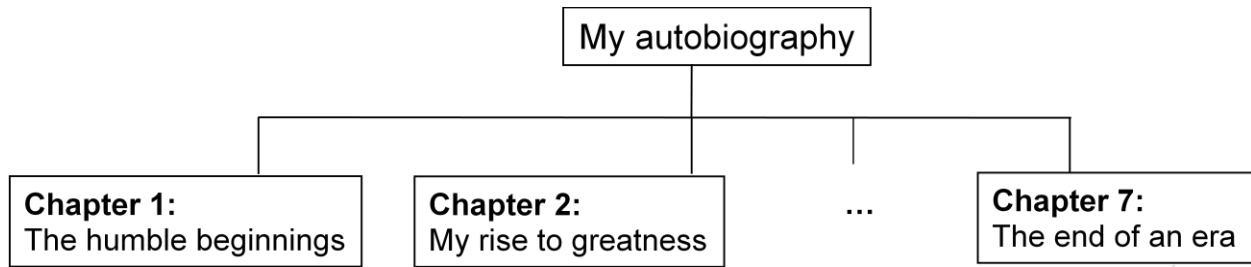


Figure 15.1: Break down of the task of writing an autobiography

2. Then we implement the solution for each part

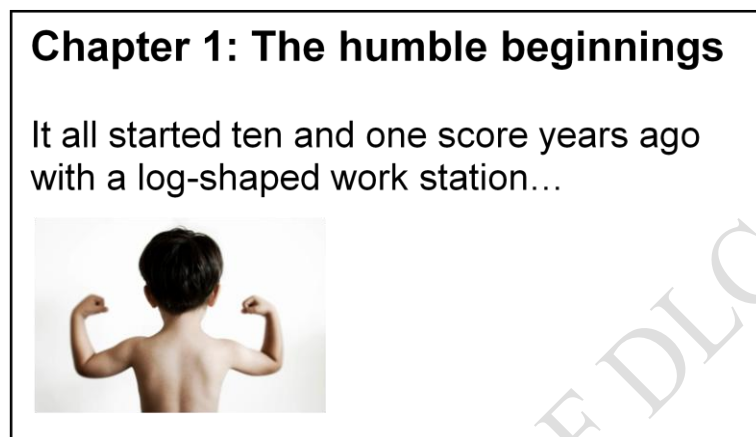


Figure 15.2: Solving a module of the task

15.1.2 Breaking a Large Problem Down

The general top down design is depicted in Figure 15.3.

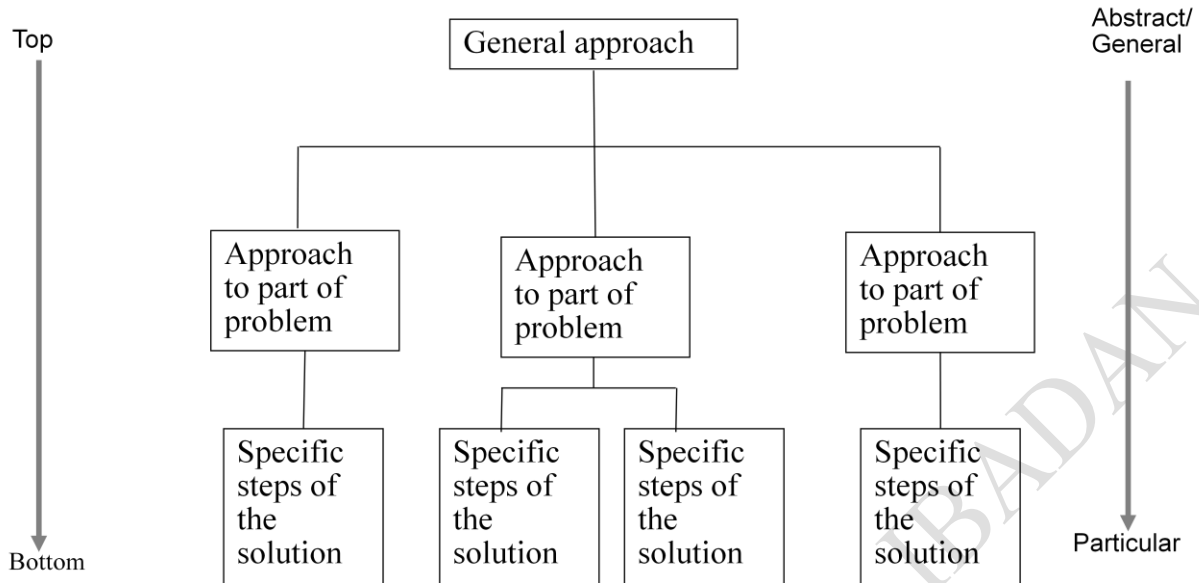


Figure 15.3: Top Down Design, extracted from Computer Science Illuminated by Dale N. and Lewis J

15.2 Procedural Programming

Procedural programming involves applying the top down approach to programming. Rather than writing a program in one large collection of instructions the program is broken down into parts. Each of these parts are implemented in the form of procedures (also called “functions” or “methods” depending upon the programming language).

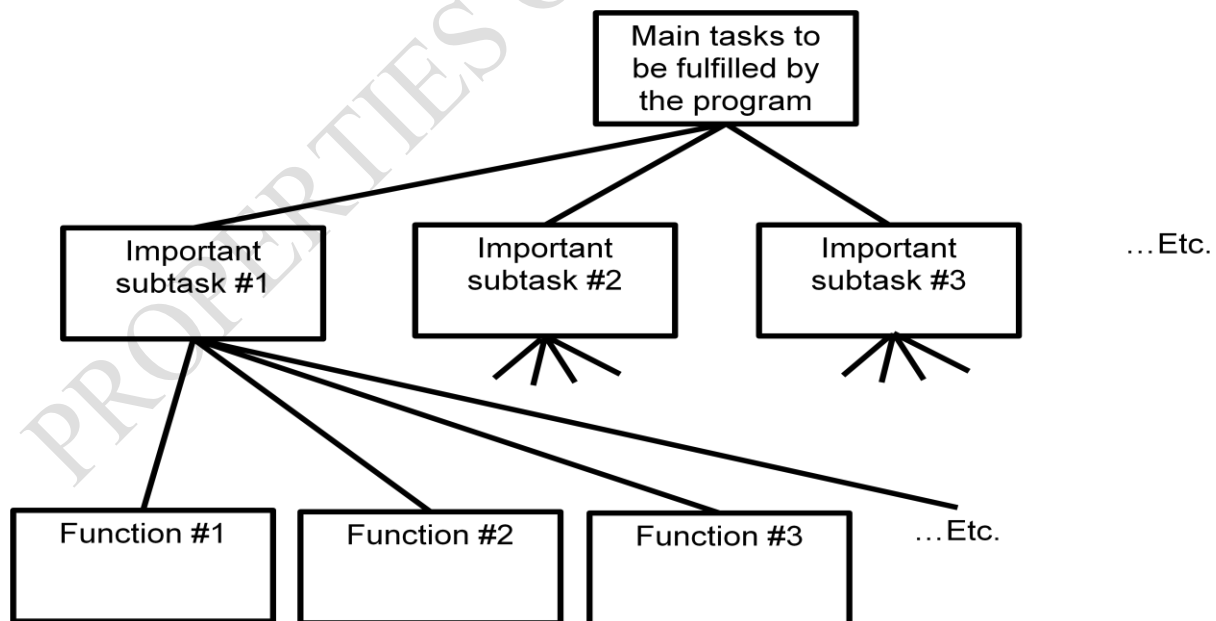


Figure 15.4: Illustrating Functions

15.3 Decomposing a Problem into Procedures

To decompose a problem into procedures:

- Break down the program by what it does (described with *actions/verbs*).
- Eventually the different parts of the program will be implemented as functions.

Example Problem

Design a program that will perform a simple interest calculation. The program should prompt the user for the appropriate values, perform the calculation and display the values onscreen.

Action/verb list in this problem would be:

- Prompt for values (Data Input)
- Calculate the Interest (Process)
- Display the output (Output)

Now draw the design.

15.4 Things Needed In Order To Use Functions

- a. Definition: Defining a function
- b. Instructions that indicate what the function will do when it runs.
- c. Call: Calling the function by a program
- d. Actually running (executing) the function.

Note: a function can be called multiple (or zero) times but it can only be defined once. Why?

15.4.1 Defining A Function

Format:

```
def <function name> ():  
    body
```

The body is the instruction or group of instructions that execute when the function executes. The rule in Python for specifying what statements are parts of the body is to use indentation

Example:

```
def displayInstructions ():  
    print ("Displaying instructions")
```

15.4.2 Calling A Function

Format:

```
<function name> ()
```

Example 1:

```
displayInstructions ()
```

Example 2:

```
def displayInstructions (): # Function Definition
    print ("Displaying instructions")
```

Main body of code (starting execution point)

```
displayInstructions() # Function Call
print ("End of program")
```

15.5 Common Mistakes with Functions in Python**(i) Functions should be defined before they can be called****Correct ☺**

```
def fun (): # Defining the function
    print ("Works")
```

main

```
fun () # Calling the function after its definition
```

Incorrect ☹

```
fun () # Calling the function before its definition
def fun (): # Defining the function
    print ("Doesn't work")
```

(ii) Forgetting the brackets during the function call

```
def fun ():
    print ("In fun")
```

Main function

```
print ("In main")
fun # The missing set of brackets does not produce a translation error
```

(iii) Indentation

Recall that in Python, indentation indicates that statements are part of the body of a function. In other programming languages, the indentation is not a mandatory part of the language but indenting is considered good style because it makes the program easier to read.

```
def main ():
print ("main") # Forgetting to indent:
```

```
main ()
```

- Inconsistent indentation:

```
def main ():
    print ("first"
    print "second")
```

```
main ()
```

(iv) Creating 'Empty' Functions

```
def fun ():
    print () # A function must have at least one statement
```

```
# Main
fun()
```

Alternative (writing an empty function: literally does nothing)

```
def fun ():
    pass
```

```
# Main
fun ()
```

15.6 Where to Create Local Variables

```
def <function name> ():
```

Local variables are created somewhere within the body of the function (indented part)

Example:

```
def fun ():
    num1 = 1
    num2 = 2
```

```
def fun ():
    num1 = 1
    num2 = 2
    print (num1, " ", num2)
```

```
# Main function
fun()
```

Variables num1 and num2 are local to function fun

Note that Local Variables Only Exist Inside a Function

```
def display ():
    print (" ")
    print ("Celsius value: ", celsius)
```

```
print ("Fahrenheit value :", fahrenheit)
```

In the above codes, what is 'celsius'??? What is 'fahrenheit'???. They are not defined in this function.

```
def convert ( ):
    celsius = float(input ("Type in the celsius temperature: "))
    fahrenheit = celsius * 9 / 5 + 32
    display ( )
```

Variables celsius and fahrenheit are local to function 'convert'

15.6.1 Solution to this kind of Problem: Parameter Passing

Variables exist only inside the memory of a function. Variables to be used inside a function that are not declared within the function must be passed as parameters to the functions.

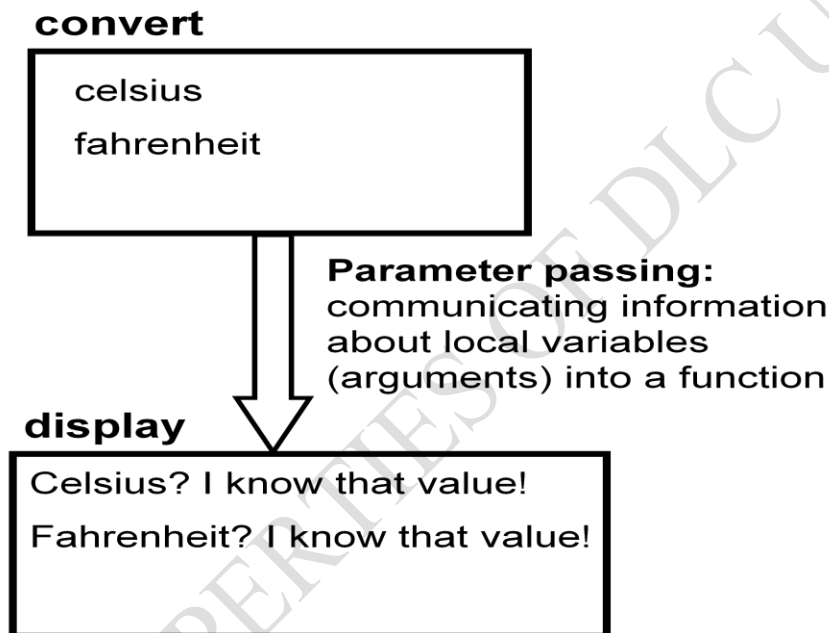


Figure 15.5: Parameter passing illustration

15.6.1.1 Parameter Passing (Function Definition)

The parameters are passed to the header of the function:

Format:

```
def <function name> (<parameter 1>, <parameter 2>...):
```

Example:

```
def display (celsius, fahrenheit):
```

15.6.1.2 Parameter Passing (Function Call)

At the point of call in the main program module, parameters are also placed in brackets.

Format:

<function name> (<parameter 1>, <parameter 2>...)

Example:

display (celsius, fahrenheit):

15.6.2 Memory and Parameter Passing

Parameters passed as arguments into functions become variables in the local memory of that function.

```
def fun (num1): # Parameter num1: local to fun
```

```
    print (num1)
    num2 = 20 # num2: local to fun
```

```
        print (num2)
num1 = 1
fun (num1) # num1: local to main
```

Full Example 1: Degree Conversion

```
def introduction ( ):
```

```
    print (" " "
    Celsius to Fahrenheit converter
```

```
    -----
    This program will convert a given Celsius temperature to an equivalent Fahrenheit value.
    " " ")
```

```
def display (celsius, fahrenheit):
```

```
    print (" ")
    print ("Celsius value: ", celsius)
    print ("Fahrenheit value:", fahrenheit)
```

```
def convert ( ):
```

```
    celsius = float(input ("Type in the celsius temperature: "))
    fahrenheit = celsius * 9 / 5 + 32
    display (celsius, fahrenheit)
```

Main function

```
def main ( ):
```

```
    introduction ( )
```



```
convert ( )  
main ( )
```

Full Example 2: Program to compute sum, mean and standard deviation of 2 numbers

```
from math import sqrt  
def add (num1, num2):  
    sum = num1 + num2  
    #print("The sum of the two numbers = ", sum)  
    return sum  
  
def average(num1, num2):  
    mean = add(num1, num2)/2  
    # print ("The mean = ", mean)  
    return mean  
  
def stdev(num1, num2):  
    print("The sum of the two numbers = ", add(num1, num2))  
    print ("The mean = ", average(num1,num2))  
    numerator = (num1 - average(num1, num2))**2 + (num2 - average(num1, num2))**2  
    std = sqrt(numerator/2)  
    #print("The standard deviation of the two numbers is ", std)  
    return std  
  
def main( ):  
    num1 = float(input("Enter the first data "))  
    num2 = float(input("Enter the second data "))  
    print("The standard deviation of the two numbers is ", stdev(num1,num2))  
  
    # add(num1, num2)  
    # average(num1, num2)  
    #stdev(num1, num2)  
  
    x = input(print("strike any key to continue"))  
    print( )  
  
main( )
```

Sample output:

```
>>>  
Enter the first data 5  
Enter the second data 9  
The sum of the two numbers = 14.0  
The mean = 7.0  
The standard deviation of the two numbers is 2.0  
strike any key to continue  
None
```

>>>

Note that the type and number of parameters must match.

Correct ☺:

```
def fun1 (num1, num2):  
    print (num1, num2)
```

```
def fun2 (num1, str1):  
    print (num1, str1)
```

main

```
def main ():  
    num1 = 1  
    num2 = 2  
    str1 = "hello"  
    fun1 (num1, num2)  
    fun2 (num1, str1)  
main ()
```

Note:

- (i) Two numeric parameters are passed into the call for 'fun1' which matches the two parameters listed in the definition for function 'fun1'
- (ii) Two parameters (a number and a string) are passed into the call for 'fun2' which matches the type for the two parameters listed in the definition for function 'fun2'

Incorrect ☹:

```
def fun1 (num1):  
    print (num1, num2)
```

```
def fun2 (num1, num2):  
    num1 = num2 + 1  
    print (num1, num2)
```

main

```
def main ():  
    num1 = 1  
    num2 = 2  
    str1 = "hello"  
    fun1 (num1, num2)  
    fun2 (num1, str1)  
main ()
```

Two numeric parameters are passed into the call for 'fun1' but only one parameter is listed in the definition for function 'fun1'

Two parameters (a number and a string) are passed into the call for 'fun2' but in the definition of the function it's expected that both parameters are numeric.

15.7 Default Parameters

Default parameters can be used to give function arguments some default values if none are provided.

Example function definition:

```
def fun (x = 1, y = 1):  
    print (x, y)
```

Example function calls (both work):

```
fun ()  
fun (2, 20)
```

15.8 Good Style: Functions

1. Each function should have one well defined task. If it doesn't then it may be a sign that it should be decomposed into multiple sub-functions.
 - a) Clear function: A function that converts lower case input to capitals.
 - b) Ambiguous function: A function that prompts for a string and then converts that string to upper case.
2. (Related to the previous point). Functions should have a self descriptive name: the name of the function should provide a clear indication to the reader what task is performed by the function.
 - a) Good: isNum, isUpper, toUpper
 - b) Bad: doIt, go
3. Try to avoid writing functions that are longer than one screen in size. Tracing functions that span multiple screens is more difficult.
4. The conventions for naming variables should also be applied in the naming of functions.
 - a) Lower case characters only.
 - b) With functions that are named using multiple words capitalize the first letter of each word but the first (most common approach) or use the underscore (less common).

15.9 Parameter Passing

What you know about scope: Parameters are used to pass the contents of variable into functions (because the variable is not in scope).

```
def fun1 ( ):
    num = 10
    fun2 (num)
```

```
def fun2 (num):
    print num
```

15.10 Scope of a Variable

The scope of an identifier (variable, constant) is where it may be accessed and used in a program module.

In Python:

- a) An identifier comes into scope (becomes visible to the program and can be used) after it has been declared.
- b) An identifier goes out of scope (no longer visible so it can no longer be used) at the end of the indented block where the identifier has been declared.

The concept of scoping applies to all programming languages. The rules for determining when identifiers come into and go out of scope will vary.

An Example

```
def fun1 ( ):
    num = 10 # 'num' comes into scope (is visible and can be used)
    # statement
    # statement
    # End of fun1 # (End of function): num goes out of scope, no longer accessible
```

```
def fun2 ( ):
    print num # Error: num is an unknown identifier, Num is no longer in scope
```

A Variant Example

```
def fun1 ( ):
    num = 10
    # statement
    # statement
    # End of fun1
```

```
def fun2 ():
    fun1 ()
    num = 20 #What happens at this point? Why?
    print num # num is accessible here, since it is declared locally within fun2
```

15.11 Global Scope

Identifiers (constants or variables) that are declared within the body of a function have a local scope (the function). Take for example,

```
def fun ():
    num = 12 # End of function fun
```

The scope of num is the function in the above code since num is declared inside the function

Now identifiers (constants or variables) that are declared outside the body of a function have a global scope (the program). For example,

```
num = 12
def fun1 ():
    # Instruction

def fun2 ():
    # Instruction

# End of program
```

The Scope of num in this case is the entire program

15.11.1 General Characteristics of Global Variables

- (i) You can access the contents of global variables anywhere in the program.
- (ii) In most programming languages you can also modify global variables anywhere as well.
 - This is why the usage of global variables is regarded as bad programming style, they can be accidentally modified anywhere in the program.
 - Changes in one part of the program can introduce unexpected side effects in another part of the program.
 - So unless you have a compelling reason you should NOT be using global variables but instead you should pass values as parameters.

15.11.2 Python Globals: Read But Not Write Access

By default global variables can be accessed globally (read access). Attempting to change the value of global variable will only create a new local variable by the same name (no write access).

```

num = 1    # Global num
def fun ( ):
    num = 2 # Local num
    print (num)

```

Prefixing the name of a variable with the keyword 'global' will indicate that all references in that function will then refer to the global variable rather than creating a local one.

```
global <variable name>
```

Example:

```

num = 1
def fun1 ( ):
    num = 2
    print (num)

```

```

def fun2 ( ):
    global num
    num = 2
    print (num)

```

```

def main ( ):
    print (num)
    fun1 ( )
    print (num)
    fun2 ( )
    print (num)

```

```
main ( )
```

15.12 Function Pre-Conditions

A pre-condition specifies things that must be true when a function is called.

Examples:

Precondition: catAge must be a non-negative number

```

def convertCatAge (catAge):
    humanAge = catAge * 7
    return humanAge

```

Precondition: y is a numeric non-zero value

```

def divide (x, y):
    z = x / y
    return z

```

15.13 Function Post-Conditions

A post-condition specifies things that must be true when a function ends.

Example:

```
def absoluteValue (number):  
    if (number < 0):  
        number = number * -1  
    return number    # Post condition: number is non-negative
```

15.14 Why Employ Problem Decomposition And Modular Design

(i) Drawback

- a) Complexity – understanding and setting up inter-function communication may appear daunting at first.
- b) Tracing the program may appear harder as execution appears to “jump” around between functions.

(ii) Benefit

- a) Solution is easier to visualize and create (decompose the problem so only one part of a time must be dealt with).
- b) Easier to test the program (testing all at once increases complexity).
- c) Easier to maintain (if functions are independent changes in one function can have a minimal impact on other functions, if the code for a function is used multiple times then updates only have to be made once).
- d) Less redundancy, smaller program size (especially if the function is used many times throughout the program).
- e) Smaller programs size: if the function is called many times rather than repeating the same code, the function need only be defined once and then can be called many times.

Summary

This Session has introduced you to the design and use of functions in programs. Modularity is an essence of modern programming practice. Now answer the following questions.

Self-Assessment Questions

- (1) What is a function? Give three advantages of using functions in a program.
- (2) Differentiate between local and global variable scopes
- (3) Write a Python program involving functions to compute the roots of a quadratic equation.

experimental set-up.

PROPERTIES OF DLC UI, IBADAN

Study Session 16: Composite Types

Expected Duration: 1 week or 2 contact hours

Introduction

Python Variables can be divided into two types: Simple (Atomic) and Aggregate (Composites). The simple variable types are integer (int), float and Boolean while the composites are strings, lists, tuples and dictionaries. We shall study the composite types in this Session.

Learning Outcomes

When you have studied this session, you should be able to explain operations on the following composite types:

- 16.1 Strings
- 16.2 Lists
- 16.3 Tuples
- 16.4 Dictionaries

16.1 Strings

Strings are just a series of characters (e.g., alpha, numeric, punctuation etc.). A string can be treated as one entity. Strings are also called texts or alphanumeric data. Examples of strings are:

- (i) "I am a boy". This is a doubly quoted string
- (ii) 'She is @ 19 Macaulay Avenue, lagos'. This is a singly quoted string

The individual elements (characters) in a string can be accessed via an index.

Note: A string with 'n' elements has an index from 0 to (n-1). This means that strings are zero-based indexed.

Consider the string below:

```
aString = "hello"

print (aString[1]) # This will print character 'e'
print (aString[4]) # This will print character ?
```

One characteristic of Python strings is that they Are Immutable. Even though it may look that a string can change they actually cannot be edited. Consider the code below and the error message received.

```
aString = "good-bye"
print (aString)
aString = "hello"
print (aString)
aString[0] = "G" # Error, cannot replace a character in a Python string
```

Sample Output

```
>>>
good-bye
hello
Traceback (most recent call last):
  File "C:/Python34/str.py", line 5, in <module>
    aString[0] = "G"
TypeError: 'str' object does not support item assignment
>>>
```

To compute the **length of a string**, use the *len*. Function. The length of a string is the total number of symbols or characters that make up that string including spaces and symbols like #, @, etc.

For example:

```
myString = input("Enter your string here --> ")
Length = len(myString)
print ("The length of the string is ", Length)
```

To join (concatenate) two or more strings together, we use the plus (+) operator.

```
# Joining two strings together
myString = input("Enter your string here --> ")
secondString = input("\nEnter the second string, Press spacebar once before typing ")
joinedString = myString + secondString
print(joinedString)
```

16.1.1 Substring Operations

Sometimes you may wish to extract out a portion of a string. E.g., To extract out “Tolu” from “Tolu Oluwaseun, Teacher”. This operation is referred to as a ‘substring’ operation in many programming languages. There are two implementations of the substring operation in Python: String slicing and splitting.

16.1.1.1 String Slicing

Slicing a string will return a portion of a string based on the indices provided. The index can indicate the start and end point of the substring.

Format:

```
string_name [start_index : end_index]
```

Example:

```
aString = "abcdefghij"
print (aString)           # This prints the entire string, abcdefghij
temp = aString [2:5]
print (temp)             # This prints cde starting from position 2, note that it stops printing at position 4
```

```
temp = aString [:5]
print (temp)           # This prints abcde starting from position 0 to 4
temp = aString [7:]
print (temp)          # This prints hij starting from position 7 to the end
```

16.1.1.2 String Splitting

Splitting a string will divide a string into portions with a particular character determining where the split occurs. The string “I am a boy” could be split into individual words “I”, “am”, “a”, “little”, “boy”.

Format:

string_name.split (<character used in the split >)

Examples:

```
aString = "man who smiles"
one, two, three = aString.split() # Default character is a space
print (one)
print (two)
print (three)
aString = "Tam, David"
last, first = aString.split(',') # comma is used to separate the aString.
print (first, last)
```

Usually, a string is split into a linear list/array; such that each word or token that makes the string occupies a location or index of the array. In the split function, we have to specify the character we used to separate the words/tokens in the string. We use space (empty character) often times. So we put empty space with apostrophes as the arguments of the split function (split(" ")). Consider the example below:

```
myString = input("Enter your string here: ")
n = len(myString.split(" "))
strArray = [ ]
for i in range(n):
    strArray.append(" ") #The last five lines creates and initialize the array with spaces

#Splitting ...

strArray = myString.split(" ")

# Printing the words/tokens
for i in range(n):
    print(strArray[i])
```

16.1.2 Strings Conceptualized as Sets

The 'in' and 'not in' operations normally used in connection with sets can be performed on a Python string.

```
passwords = "aaa abc password xxx"
password = input ("Password: ")
if password in passwords:
    print "You entered an existing password, enter a new one"
```

To loop (iterate) through the elements, *for* loop can be used:

```
sentence = "hihi there!"
for temp in sentence:
    sys.stdout.write(temp)
```

Note: Use of the write function requires the 'import' of the sys library. The library allows for more precise formatting than the standard print

```
import sys
# Rest of your program
```

A string with a number of repeated characters can be initialized in a number of ways.

```
aString = "xxxxxxx"
aString = "hi!" * 5
```

16.1.3 String Testing Functions

These functions test a string to see if a given condition has been met and return either "True" or "False" (Boolean). These functions will return false if the string is empty (less than one character).

Format:

string_name.function_name ()

Boolean Function	Description
isalpha ()	Only true if the string consists only of alphabetic characters.
isdigit ()	Only returns true if the string consists only of digits.

isalnum ()	Only returns true if the string is composed only of alphabetic characters or numeric digits.
islower ()	Only returns true if the alphabetic characters in the string are all lower case.
isspace ()	Only returns true if string consists only of whitespace characters (“ “, “\n”, “\t”)
isupper ()	Only returns true if the alphabetic characters in the string are all upper case.

16.1.4 Applying A String Testing Function

```

ok = False
while (ok == False):
    temp = input ("Enter numbers not characters: ")
    ok = temp.isdigit()
    if (ok == False):
        print(temp, "is not a number")
    else:
        print("done, it is purely a number")
num = int (temp) # To parse the number to integer value
num = num + num
print(num)

```

Sample Output:

```

>>>
Enter numbers not characters: 78
Done, it is purely a number
156
>>>

```

16.1.5 Functions That Modify Strings

These functions return a modified version of an existing string (leaves the original string intact).

Function	Description
<code>lower ()</code>	Returns a copy of the string with all the alpha characters as lower case (non-alpha characters are unaffected).
<code>upper ()</code>	Returns a copy of the string with all the alpha characters as upper case (non-alpha characters are unaffected).
<code>strip ()</code>	Returns a copy of the string with all leading and trailing whitespace characters removed.
<code>lstrip ()</code>	Returns a copy of the string with all leading (left) whitespace characters removed.
<code>rstrip ()</code>	Returns a copy of the string with all trailing (right) whitespace characters removed.
<code>lstrip (char)</code>	Returns a copy of the string with all leading instances of the character parameter removed.
<code>rstrip (char)</code>	Returns a copy of the string with all trailing instances of the character parameter removed.

16.1.5.1 Example Uses of Functions That Modify Strings

```
myString = "here I aM"  
print(myString)  
myString = myString.upper ()  
print(myString)  
myString = "\n\nHere I am"  
print(myString)  
myString = myString.lstrip ('n')  
print(myString)  
myString = "xxI am yxur frxenx "  
myString = myString.lstrip ('x')  
print(myString)
```

Output:

```
>>>  
here I aM  
HERE I AM  
\n\nHere I am  
Here I am  
I am yxur frxenx
```

Example: String Slicing

```
aString = "abcdefghij"
print (aString)
temp = aString [2:5] #Extracts 3 (5 – 2) characters from index 2 (index 3 in real life)
print (temp)
temp = aString [:5] #Extracts characters from index 0 to 4, five characters from beginning
print (temp)
temp = aString [7:] # Extracts characters from index 7 (8 indeed) to the end
print (temp)
```

Output:

```
>>>
abcdefghij
cde
abcde
hij
>>>
```

16.1.6 Functions to Search Strings

Function	Description
endswith (substring)	A substring is the parameter and the function returns true only if the string ends with the substring.
startswith (substring)	A substring is the parameter and the function returns true only if the string starts with the substring.
find (substring)	A substring is the parameter and the function returns the lowest index in the string where the substring is found (or -1 if the substring was not found).
replace (oldstring, newstring)	The function returns a copy of the string with all instances of 'oldstring' replace by 'newstring'

16.1.6.1 Examples of Functions to Search Strings

Example 1: Find

```
mystring = "Sunday is a boy"
a = mystring.find("is a")
print (a)
```

Sample Output

```
>>>
7
>>>
```

Example 2: endsWith

```
temp = input ("Enter a sentence: ")
if not ((temp.endswith('.') or (temp.endswith('!')) or (temp.endswith('?'))):
    print("Not a sentence")
```

Output:

```
>>>
Enter a sentence: I am a big man.
I am a big man. is a sentence
>>>
```

Example 3: To find and replace

```
theString = "XXabcXabcabc"
print("Given the string: ", theString)
index = theString.find("abc")
print("abc is found at beginning index ", index)
theString = theString.replace("abc", "Abc")
print("Replacing 'abc' with 'ABC' gives: ", theString)
```

Output:

```
>>>
Given the string: XXabcXabcabc
abc is found at beginning index 2
Replacing 'abc' with 'ABC' gives: XXABCXABCABC
>>>
```

16.2 Lists

In many programming languages a list is implemented as an array. Python lists have many of the characteristics of the arrays in other programming languages but they also have many other features. Arrays are sets or lists of homogenous values, stored contiguously in memory.

Suppose we are confronted with solving a problem depicted below.

Write a program that will track the percentage grades for a class of students. The program should allow the user to enter the grade for each student. Then it will display the grades for the whole class along with the average.

Suppose we have a large number of student, say 100, how do we handle the problem. One way is suggested below for a class of 5 students:

```
CLASS_SIZE = 5
stu1 = float(input("Enter grade for student no. 1: "))
stu2 = float(input("Enter grade for student no. 2: "))
stu3 = float(input("Enter grade for student no. 3: "))
stu4 = float(input("Enter grade for student no. 4: "))
stu5 = float(input("Enter grade for student no. 5: "))
total = stu1 + stu2 + stu3 + stu4 + stu5
average = total / CLASS_SIZE
print( )
print("GRADES")
print("The average grade is %.2f%%", %average)
print("Student no. 1: %.2f", %stu1)
print("Student no. 2: %.2f", %stu2)
print("Student no. 3: %.2f", %stu3)
print("Student no. 4: %.2f", %stu4)
print("Student no. 5: %.2f", %stu5)
```

This approach adopted in this program is not efficient!

What were the problems with the previous approach?

It is full of redundant statements. Yet a loop could not be easily employed given the types of variables that you have seen so far.

So, What's Needed?

A composite variable that is a collection of another type. The composite variable can be manipulated and passed throughout the program as a single entity. At the same time each element can be accessed individually. What's needed...a list!

16.2.1 Creating A List (No Looping)

This step is mandatory in order to allocate memory for the list. Omitting this step (or the equivalent) will result in a syntax error.

Format:

```
<list_name> = [<value 1>, <value 2>, ... <value n>]
```

Example:

```
percentages = [50.0, 100.0, 78.5, 99.9, 65.1]
letters = ['A', 'B', 'A']
names = ["Nwosu Isioma", "Audu Usman", "Akinola Solomon"]
```

To access an element of a list, state the list name with the index of the element to access in square brackets. For instance, from the above example, percentages [1] = 100.0. In list too, indexing of elements starts from zero.

16.2.2 Creating A List (With Loops)

Step 1: Create a variable that is a reference to the list

Format:

```
<list name> = [ ]
```

Example:

```
classGrades = [ ]
```

Step 2: Initialize the list with the elements

General format:

Within the body of a loop create each element and then append the new element on the end of the list.

Example:

```
for i in range (0, 5, 1):  
    classGrades.append (0)
```

Example Program 1: Computing Standard Deviation of some numbers

```
print ("This program computes the mean and standard deviation of some data")  
n = int(input("Enter the total number of data"))  
#Declaring the array or list  
score = [ ]  
for i in range (0, n, 1):  
    score.append(0)  
  
#Appending data into the list  
for i in range (0, n, 1):  
    # k = i + 1  
    score[i] = float(input ("Enter score for student no. %d>" %(i+1)))  
  
#Summing the data altogether  
sum = 0  
for i in range (0, n, 1):  
    sum += score[i]  
  
#Computing the Mean  
mean = sum/n
```

```

from math import sqrt
#Computing the standard deviation
var = 0
for i in range (0, n, 1):
    var += ((score[i] - mean)**2)/n

stdev = sqrt(var)

print("Sum of data = ", sum)
print("The mean = ", mean)
print ("Standard Deviation = ", stdev)

```

Sample Output:

```

>>>
This program computes the mean and standard deviation of some data
Enter the total number of data 5
Enter score for student no. 1> 9
Enter score for student no. 2> 5
Enter score for student no. 3> 7
Enter score for student no. 4> 6
Enter score for student no. 5> 4
Sum of data = 31.0
The mean = 6.2
Standard Deviation = 1.7204650534085253
>>>

```

Example Program 2: Using functions

```

CLASS_SIZE = int(input("Enter total number of data to process here: "))

def read(classGrades):
    total = 0

    for i in range (0, CLASS_SIZE, 1):
        # Because list indices start at zero add one to the student number.
        temp = i + 1
        #print("Enter grade for student no.", temp, ":")
        classGrades[i] = float(input ("Enter grade for student no. %d>" %temp))
        total = total + classGrades[i]
        average = total / CLASS_SIZE
    return (classGrades, average)

def display(classGrades, average):
    print( )
    print("GRADES")
    print("The average grade is %.2f%%" %average)

```

```

for i in range (0, CLASS_SIZE, 1):
    # Because array indices start at zero add one to the student number.
    temp = i + 1
    print("Student No. %d: %.2f%%" %(temp,classGrades[i]))

# Computing Standard Deviation
from math import sqrt

def std_dev(classGrades, average):
    var = 0
    for i in range(0, CLASS_SIZE, 1): #Computing variance
        var += ((classGrades[i] - average)**2)/CLASS_SIZE

    # Computing the standard deviation
    std = sqrt(var)
    print("\n\nStandard Deviation = %.2f" %std)

def main():
    classGrades = [ ]
    for i in range (0, CLASS_SIZE, 1):
        classGrades.append(0)

    classGrades, average = read (classGrades)
    display (classGrades, average)
    std_dev(classGrades, average)

main ( )

```

Sample Output:

```

>>>
Enter total number of data to process here: 5
Enter grade for student no. 1>56
Enter grade for student no. 2>62
Enter grade for student no. 3>35
Enter grade for student no. 4>48
Enter grade for student no. 5>79

```

GRADES

```

The average grade is 56.00%
Student No. 1: 56.00%
Student No. 2: 62.00%
Student No. 3: 35.00%
Student No. 4: 48.00%
Student No. 5: 79.00%

```

Standard Deviation = 14.63

>>>

16.2.3 Printing Lists

Although the previous examples stepped through each element of the list in order to display its contents onscreen. However, if you want to quickly check the contents (and not worry about details like formatting) then you can simply use a print statement as you would with any other variable.

Example:

```
print (classGrades)
```

Output:

```
[10, 20, 30, 40, 50]
```

Take Care Not to Exceed the Bounds of the List!

```
list = [0, 1, 2, 3]
for i in range (0, 4, 1):
    print (list [i])
print ()
print (list [4]) ← ???    #This is list or array index-out-of-bound
```

One way of avoiding an overflow of the list is to use a constant in conjunction with the list.

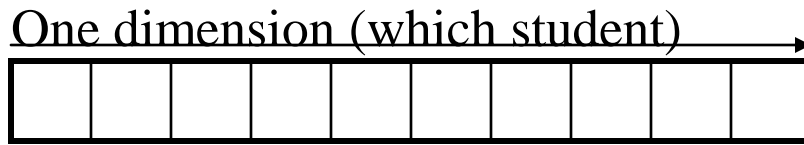
```
SIZE = 100
```

The value in the constant controls traversals of the list:

```
for i in range (0, SIZE, 1):
    myList [i] = int(input ("Enter a value:" ))
for i in range (0, SIZE, 1):
    print (myList [i])
```

16.2.4 Copying Lists

A list variable is not actually a list! Instead the list variable is actually a reference to the list. This is important because if you use the assignment operator to copy from list to another you will end up with only one list.

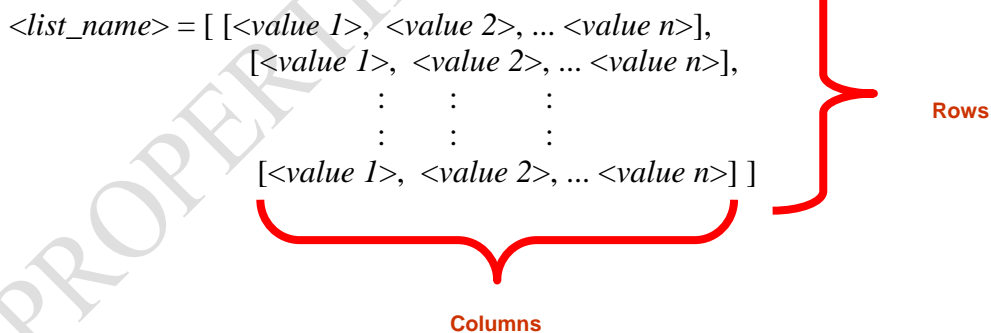


2. Two Dimensional (2D) list

Let us expand the grades program. Again there is one dimension that specifies which student's grades are being accessed. The other dimension can be used to specify the lecture section. Notice that each row is merely a 1D list. This means a 2D list is a list containing rows of 1D lists. List elements are specified in the order of [row] [column].

	First student	Second student	Third student	...
L01	[Row 0][Col 0]			
L02				
L03			[Row 2][Col 2]	
L04				
L05		[Row 4][Col 1]		
:				
L0N				

16.2.6 Creating and Initializing a Multi-Dimensional List in Python: General structure



```

matrix = [ [0, 0, 0],
            [1, 1, 1],
            [2, 2, 2],
            [3, 3, 3]]
for r in range(0, 4, 1):
    print (matrix[r])
  
```

```

for r in range (0,4, 1):
    for c in range (0,3,1):
        sys.stdout.write(str(matrix[r][c]))
    print()

```

16.2.6.1 General structure (Using loops):

Create a variable that refers to a 1D list. The outer loop traverses the rows. Each iteration of the outer loop creates a new 1D list. Then the inner loop traverses the columns of the newly created 1D list creating and initializing each element in a fashion similar to how a single 1D list was created and initialized.

Example (Using loops):

```

aGrid = [ ]           # Create a reference to the list
for r in range (0, 3, 1): # Outer loop runs once for each row
    aGrid.append ([ ])  # Create a row (a 1D list)
    for c in range (0, 3, 1): # Inner loop runs once for each column
        aGrid[r].append (" ") # Create and initialize each element (1D list)

```

Example 2D List Program: A Character-Based Grid

```

import sys
aGrid = [ ]
aGrid = [ ]
for r in range (0, 2, 1):
    aGrid.append ([ ])
    for c in range (0,3,1):
        aGrid[r].append (str(r+c))

for r in range (0,2,1):
    for c in range (0,3,1):
        sys.stdout.write(str(aGrid[r][c]))
    print()

```

Output:

```

>>>
0 1 2
1 2 3
>>>

```


Another Example: Manipulating 2-dimensional list

```
nrows = int(input("Enter number of rows"))
ncols = int(input("Enter number of columns"))

matrix = [ ]          # Create a reference to the list
for r in range (nrows): # Outer loop runs once for each row
    matrix.append ([ ]) # Create a row (a 1D list)
    for c in range (ncols): # Inner loop runs once for each column
        matrix[r].append (" ") # Create and initialize each element (1D list)

#Read in data into the two dimensional matrix

for r in range(nrows):
    for c in range(ncols):
        matrix[r][c] = float(input("Enter data at row %d column %d" %((r+1),(c+ 1))))

#Printing the original matrix
print("\n\nThe original matrix ...\n")
print(matrix)

# Computing the mean value of all data in the matrix
sum = 0
for r in range(nrows):
    for c in range(ncols):
        sum += matrix[r][c]

mean = sum/(nrows*ncols)

print("Mean value of data in the matrix = %.2f " %mean)

# Computing the maximum value of all data in the matrix
maxim = matrix[0][0]
rpos = 0
cpos = 0
for r in range(nrows):
    for c in range(ncols):
        if (matrix[r][c] > maxim):
            maxim = matrix[r][c]
            rpos = r+1
            cpos = c +1

print("Maximum value of data in the matrix = %.2f" %maxim)
#print("The maximum occurs at row %d column %d" %(rpos, cpos))
```

```

for r in range(nrows):
    for c in range(ncols):
        if (matrix[r][c] == maxim):
            print("The maximum occurs at row %d column %d" %(r+1, c+1))

```

Sample Output

```

>>>
Enter number of rows 3
Enter number of columns 4
Enter data at row 1 column 1 2
Enter data at row 1 column 2 3
Enter data at row 1 column 3 5
Enter data at row 1 column 4 6
Enter data at row 2 column 1 4
Enter data at row 2 column 2 8
Enter data at row 2 column 3 4
Enter data at row 2 column 4 2
Enter data at row 3 column 1 1
Enter data at row 3 column 2 6
Enter data at row 3 column 3 8
Enter data at row 3 column 4 1

The original matrix ...

[[2.0, 3.0, 5.0, 6.0], [4.0, 8.0, 4.0, 2.0], [1.0, 6.0, 8.0, 1.0]]
Mean value of data in the matrix = 4.17
Maximum value of data in the matrix = 8.00
The maximum occurs at row 2 column 2
The maximum occurs at row 3 column 3
>>>

```

16.2.7 Heterogeneous List

What if different types of information need to be tracked in the list? List elements need not store the same data type.

Example, storing information about a client:

The client's data may look like below:

Name	...series of characters
Phone number	...numerical or character
Email address	...series of characters
Total purchases made	...numerical or character

If just a few clients need to be tracked, then a simple list can be employed:

```
firstClient = ["James Tam",
              "(403)210-9455",
              "tamj@cpsc.ucalgary.ca",
              0]
```

Or as a small example:

```
def display (firstClient):
    print "DISPLAYING CLIENT INFORMATION"
    print "-----"
    for i in range (0, 4, 1):
        print firstClient [i]
```

MAIN

```
firstClient = ["James Tam"
              "(403)210-9455",
              "tamj@cpsc.ucalgary.ca",
              0]
display (firstClient)
```

If only a few instances of the composite type (e.g., “Clients”) need to be created then multiple instances single lists can be employed.

```
firstClient = ["Olu James"
              "(234) 8126735426",
              "oluj@yahoo.com",
              20]

secondClient = ["Usman Nwosu"
               "(234) 9056787352",
               "usmawos@gmail.com",
               100]
```

16.2.8 Some List Operations (1)

Operation name	Operator	Description
Indexing	[]	Access a list element
Concatenation	+	Combine lists
Repetition	*	Concatenate a repeated number of times
Membership	in	Query whether an item is a member of a list

Membership	not in	Query whether an item is not a member of a list
Length	len	Return the number of items in a list
Slicing	[:]	Extract a part of a list

Examples: Concatenation and Repetition

```
list1 = [1, 2.0, "foo"]
list2 = [[1,2,3], "bar"]
print (list1)
print (list2)
list1 = list1 * 2
print (list1)
list3 = list1 + list2
print (list3)
```

Output:

```
>>>
[1, 2.0, 'foo']
[[1, 2, 3], 'bar']
[1, 2.0, 'foo', 1, 2.0, 'foo']
[1, 2.0, 'foo', 1, 2.0, 'foo', [1, 2, 3], 'bar']
>>>
```

Examples: Membership

```
print("Example 1: ")
recall_list = ["vpn123", "NCC-75633", "gst7"]
item = input ("Product code to check for recall: ")
if item in recall_list:
    print("Your product was on the recall list, take it back")
else:
    print("You're safe")
print()
print("Example 2:")
days = ["Sun", "Mon", "Tue", "Wed", "Thur", "Fri", "Sat"]
temp = input("Press any key to see the days in a week ")
for temp in days:
    print(temp)
```

Output:

```
>>>
Example 1:
Product code to check for recall: vpn123
Your product was on the recall list, take it back
```

Example 2:

Press any key to see the days in a week

Sun

Mon

Tue

Wed

Thur

Fri

Sat

>>>

16.2.9 Some Useful List Operations (2)

Operation	Format	Description
Append	<code>list_name.append (item)</code>	Adds a new item to the end of the list
Insert	<code>list_name.insert (i, item)</code>	Inserts a new item at index 'i'
Sort	<code>list_name.sort ()</code>	Sorts from smallest to largest
Reverse	<code>list_name.reverse ()</code>	Reverses the current order of the list
Count	<code>list_name.count (item)</code>	Counts and returns the number of occurrences of the item

16.3 Tuples

Much like a list, a tuple is a composite type whose elements can consist of any other type. Tuples support many of the same operators as lists such as indexing. However, tuples are immutable. Tuples are used to store data that should not change.

16.3.1 Creating Tuples

Format:

`tuple_name = (value1, value2...valuen)`

Example:

`tup = (1,2,"foo",0.3)` # Note, normal brackets are used for tuples

A Small Example Using Tuples

`tup = (1, 2,"foo",0.3)`

```
print (tup)
print (tup[2])
tup[2] = "bar" ← Error: "TypeError: object does not support item assignment"
```

Although it appears that functions in Python can return multiple values, they are in fact consistent with how functions are defined in other programming languages. Functions can either return zero or *exactly one value* only. Specifying the return value with brackets merely returns one tuple back to the caller.

```
def fun ():
    return (1, 2, 3) # Returns a tuple with three elements
def fun (num):
    if (num > 0):
        print "pos"
        return
    elif (num < 0):
        print "neg"
    return # Nothing is returned back to the caller
```

16.4 Dictionaries

A dictionary is a special purpose composite type that maps keys (which can be any immutable type) to a value (like lists it can be any value). The keys can be used to later lookup information about the value e.g., looking up the definition for a word in a dictionary.

16.4.1 Creating A Small Dictionary

Format: Defining the entire dictionary all at once

`<dictionary_name> = {key1:value1, key2:value2...keyn:valuen}` #It's all about key-value pair

Example: Defining the entire dictionary all at once

```
dict = {"one": "yut", "two": "yee", "three": "saam"}
```

16.4.2 Creating A Large Dictionary

Format:

```
dictionary_name = { }
dictionary_name [key1] = value1
dictionary_name [key2] = value2
:           :           :
dictionary_name [keyn] = valuen
```

Example:

```
dict = { }
dict ["word1"] = ["Dictionary definition for word1"]
dict ["word2"] = ["Dictionary definition for word2"]
```

Examples of Creating Dictionaries

```
dict = { }
dict ["GIS"] = ["GIS is Geography Information System"]
dict ["Geog"] = ["Geography, the study of Earth and its features"]
dict ["Feature"] = ["Physical outlook or characteristics of an object"]
```

```
temp = input ("Enter the word to define from GIS or Geog: ")
if temp in (dict):
    print(dict[temp])
else:
    print ("Word not found in the dictionary")
```

Sample Output

```
>>>
Enter the word to define from GIS or Geog: you
Word not found in the dictionary
>>>
=====RESTART=====
>>>
Enter the word to define from GIS or Geog: GIS
['GIS is Geography Information System']
>>>
```

16.4.3 Removing Dictionary Entries**Format:**

```
del <dictionary_name> [key]
```

Example:

```
del dict ["one"]
```

Example: Deletion And Checking For Membership

```
dict = { }
dict ["one"] = "Sentence one"
dict ["two"] = "Sentence two"
dict ["three"] = "Sentence three"
if "one" in dict:
    print("key one is in the dictionary")
del dict["one"]
if "one" not in dict:
```

```
print("key one is NOT in the dictionary")
```

Output:

```
>>>  
key one is in the dictionary  
key one is NOT in the dictionary  
>>>
```

Summary

You should now know:

- (i) The difference between a mutable and an immutable type
- (ii) How strings are actually a composite type
- (iii) Common string functions and operations
- (iv) Why and when a list should be used
- (v) How to create and initialize a list
- (vi) How to access or change the elements of a list
- (vii) Copying lists: How does it work/How to do it properly
- (viii) When to use lists of different dimensions
- (ix) How to use the 'in' operator in conjunction with lists
- (x) How a list can be used to store different types of information (non-homogeneous composite type)
- (xi) Common list operations and functions
- (xii) How to define an arbitrary composite type using a class
- (xiii) What a tuple is and how do they differ from other composite types
- (xiv) How to create a tuple and access the elements
- (xv) Why functions at most return a single value
- (xvi) What a dictionary is and when can they can be used
- (xvii) How to create a dictionary, access and remove elements

Self Assessment Questions

1. Write a program to compute the mean deviation of n integers, n is an integer
2. Create a dictionary of some keywords in Computer Science and let your program be useful to look for any word in the dictionary.
3. In a JAMB's Matriculation Examination, there are n number of candidates offering Mathematics, English Language, Physics and Chemistry to read Computer Science at Redeemer University. The cut-off point is 210 for the Session. Assuming the data is stored in a two-dimensional list, write a Python program to compute
 - (a) The sum of the scores obtained by the candidates in the examination
 - (b) The students that gained admission in that Session
 - (c) The total number of candidates that gained the admission

4. (a) What are Python *Strings*? What are the characteristics of Python *Strings*?
- (b) Using a sample string of your own, write Python codes to
 - (i) Read in a string and write out its characters one by one.
 - (ii) Print out the first 5 characters from the string.
 - (iii) Concatenate two strings together.
 - (iv) Split a string into its constituent words

PROPERTIES OF DLC UI, IBADAN

Further References

Algorithm, Flowcharts and Pseudocodes:

www.tud.ttu.ee/im/Msury.Mahunnah/.../algorithm_flowchart_pseudocode.docx

Computer Fundamentals:

http://www.tutorialspoint.com/computer_fundamentals/computer_quick_guide.htm

Computer Hardware: http://en.wikipedia.org/wiki/Computer_hardware

Folajimi, Y, Makolo, A. and Oguntunde, T. (2012) Python Lecture Notes, Department of Computer Science, University of Ibadan.

<https://cs.calvin.edu/activities/books/processing/text/01computing.pdf>

Introduction to Computer Science:

https://en.wikiversity.org/wiki/Introduction_to_Computer_Science

Introduction to computers:

http://www.just.edu.jo/~mqais/CIS99/PDF/Ch.01_Introduction_%20to_computers.pdf

James Tam, Lecture Notes on Python, Computer Science Department, University of Calgary.

Number Systems:

https://www.cs.princeton.edu/courses/archive/spr15/cos217/lectures/03_NumberSystems.pdf

Ojo, S. O. (1990). Introduction to Computers, Revised Edition, Lecture notes, Department of Computer Science, University of Ibadan.

Osofisan A. O. and Akinkunmi B. O. (2012). The Science of Computing, Book Chapter in GES 104 Textbook, Science and Mankind.

Vermaat, Misty E. Microsoft Office 2013 Introductory. Cengage Learning, p.IT3. 2014