Key Dynamics in Computer Programming

Edited by: **Adele Kuzmiakova**

AP | ARCLER PRESS

Key Dynamics in Computer Programming

KEY DYNAMICS IN COMPUTER PROGRAMMING

Edited by:

Adele Kuzmiakova

ARCLER PRESS

www.arclerpress.com

Key Dynamics in Computer Programming

Adele Kuzmiakova

Arcler Press
224 Shoreacres Road
Burlington, ON L7L 2H2
Canada
www.arclerpress.com
Email: orders@arclereducation.com

e-book Edition 2023

ISBN: 978-1-77469-638-5 (e-book)

ISBN: 978-1-77469-438-1 (Hardcover)

ABOUT THE EDITOR

Adele Kuzmiakova is a machine learning engineer focusing on solving problems in machine learning, deep learning, and computer vision. Adele currently works as a senior machine learning engineer at Ifolor focusing on creating engaging photo stories and products. Adele attended Cornell University in New York, United States for her undergraduate studies. She studied engineering with a focus on applied math. Some of the deep learning problems Adele worked on include predicting air quality from public webcams, developing a real-time human movement tracking, and using 3D computer vision to create 3D avatars from selfies in order to bring online clothes shopping closer to reality. She is also passionate about exchanging ideas and inspiring other people and acted as a workshop organizer at Women in Data Science conference in Geneva, Switzerland.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AI	artificial intelligence
ALU	arithmetic logic unit
ANSI	American National Standards Institute
BAL	basic assembly language
CBOL	common commercial-oriented language
CPUs	computer processing units
DD	domain-dependent
DDEX	DD-software experimental
DMA	direct memory access
EEPROM	electronically erasable programmable read-only memory
GUI	graphical user interface
ISO	International Standards Organization
ISVs	independent software vendors
MSB	most significant bit
OOP	object-oriented programming
OS	operating system
PC	personal computers
PWA	progressive web app
ROM	read-only memory
SSEC	electronic selective sequence calculator
UWP	universal windows app

PREFACE

A program is created by first defining a task and then expressing it in a computer language that is appropriate for the application. The specification is then converted into a coded program that can be directly executed by the machine on which the task is to be performed, usually in numerous steps. Machine language refers to the coded program, whereas problem-oriented languages refer to languages that are ideal for original formulation. C, Python, and C++ are only a few of the many problem-solving languages that have been invented.

Computers come with a variety of programs that are meant to help users do jobs and improve system performance. The operating system (OS), which is a collection of programs, is as crucial to the operation of a computer system as its hardware. Current technology allows some operating features to be built into a computer's central processing unit as fixed programs (introduced by client orders) at the time of production. The operating system may have control over user programs during execution, such as when a time-sharing monitor suspends one program and activates another, or when a user program is begun or terminated, such as when a scheduling software chooses which user program will be executed next. Certain operating-system programs, on the other hand, run as stand-alone modules to make the programming process easier. While translators (assemblers or compilers) convert an entire program from one language to another, interpreters execute a program sequentially. Interpreters translate at each step and debuggers execute a program piecemeal and monitor various circumstances, allowing the program to check whether the program's operation is correct or not.

This book aims to help the student understand computer programming by presenting the fundamentals of computer hardware and software, computer programs, operating systems, major programming languages, and an introduction to Windows operating systems.

Chapter 1 introduces the readers to the fundamentals of computers and computer programs. Chapter 2 deals with the classification of computer programs. Chapter 3 discusses the fundamentals of programming languages. Chapters 4 and 5 introduce the readers to two major languages: Python and C language.

Chapter 6 illustrates the idea of dynamic programming and its uses. Chapter 7 focuses on the fundamentals of different operating systems. Finally, Chapter 8 deals with the timeline of Windows with a focus on its features.

We have not hesitated to be prescriptive: to claim that accumulated experience shows that certain constructs are to be preferred, and others to be avoided or at least used with caution Of course, any book on programming languages should not be taken as a reference manual for any particular language. The book equips you with insights so that you can learn to analyze languages and not to study the peculiarities of any language in depth. Nor is the book a guide to the choice of a language for any particular project. The goal is to supply the student with the conceptual tools needed to make such a decision.

CHAPTER 1

FUNDAMENTALS OF COMPUTERS AND PROGRAMMING

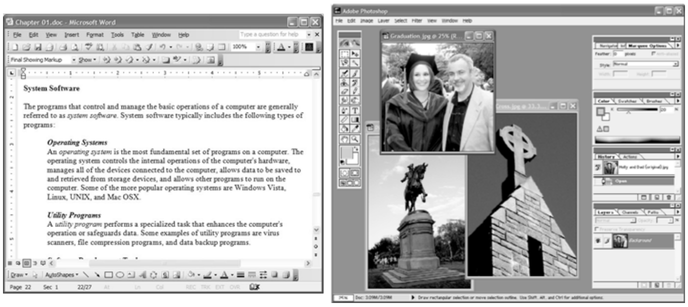
CONTENTS

1.1. INTRODUCTION

Take a look at various different approaches people utilize computers. In school, students employ computers for sending emails, looking for articles, taking online classes, and writing papers. Computers are used in the workplace to analyze data, generate presentations, perform business transactions, interact with clients and coworkers, and drive machinery in industrial plants, among other things. People use computers at home to pay bills, shop online, communicate with family and friends, and play video games. Car navigation systems, iPods®, cell phones, and a variety of other devices are all computer devices. Computers have nearly unlimited applications in our daily lives (Liu, 2020).

Because computers can be programmed, they can do a wide range of tasks. This means that computers are not meant to make a single task but rather to perform any task that their programs instruct them to perform. A program is a set of instructions afterward a computer to complete a task. Figure 1.1, for example, depicts interfaces from two widely used programs: Adobe Photoshop and Microsoft Word. Microsoft Word is a word processing tool that lets you use your computer to generate, modify, and print documents. Adobe Photoshop is a visual image editing tool that lets you work with photos captured with your digital camera (Feurzeig et al., 1970).

The term “software” refers to computer programs. The software on a computer is critical since it uses everything the machine does. All the software we employ to make our computers usable is created by people who work as software developers or programmers. A programmer, also known as a software developer, is a person who has completed the necessary training and acquired the necessary abilities to design, create, and test computer programs. Computer programming is an interesting and fulfilling professional path (Knuth and Pardo, 1980). Programmers are employed in a wide range of fields today, including business, medical, agricultural, law enforcement, government, academia, entertainment, and many more.


Chapter 01.doc - Microsoft Word
File Edit View Insert Format Tools Table Window Help
Adobe Photoshop
Graduation.jpg @ 25%

System Software

The programs that control and manage the basic operations of a computer are generally referred to as *system software*. System software typically includes the following types of programs:

Operating Systems
An *operating system* is the most fundamental set of programs on a computer. The operating system controls the internal operations of the computer's hardware, manages all of the devices connected to the computer, allows data to be saved to and retrieved from storage devices, and allows other programs to run on the computer. Some of the more popular operating systems are Windows Vista, Linux, UNIX, and Mac OSX.

Utility Programs
A *utility program* performs a specialized task that enhances the computer's operation or safeguards data. Some examples of utility programs are virus scanners, file compression programs, and data backup programs.

Figure 1.1. An image editing software and a text processing program.

Source: https://www.fiverr.com/broewnis/convert-adobe-photoshop-to-microsoft-word.

Python is the programming language that is used in this book to expose you to the basic ideas of computer programming. Before we can begin to explore those notions, you must first understand some fundamental principles about computers and how they function. This chapter will provide you with a firm basis of understanding that you will be able to draw on throughout your computer science studies. First, we will go through the physical components that are often used in the construction of computers. Following that, we will see how computers store data and run programs. Ultimately, we will get a brief overview of the Python programming language and the software that you will need to create Python programs (Horn et al., 2009).

The physical devices that make up a computer are referred to as the computer's hardware in this context. Software is the term used to describe the programs that operate on a computer.

1.2. HARDWARE

In computing, the phrase "hardware" indicates all the physical components or devices that make up a computer's structure. A computer is not a separate device but rather a collection of devices that all operate at the same time to form a system. Each device in a computer is like the different instruments in a symphony orchestra in that each device has a specific function (Tejada et al., 2001).

For anyone who has done any computer shopping, you have probably seen sales narratives listing elements like microprocessors, memory, graphics cards, video displays, hard disc drives, etc. Microprocessors, memory, and hard disc drives are just a few of the components that can be found in a computer. Without prior computer knowledge or at least a buddy who is knowledgeable about computers, it may be difficult to comprehend what each of these separate components performs on its own. As shown in Figure 1.2, a distinctive computer system is comprised of the following major components (So and Brodersen, 2008):

- CPU;
- Secondary storage devices;
- Main memory;
- Output devices;
- Input devices.

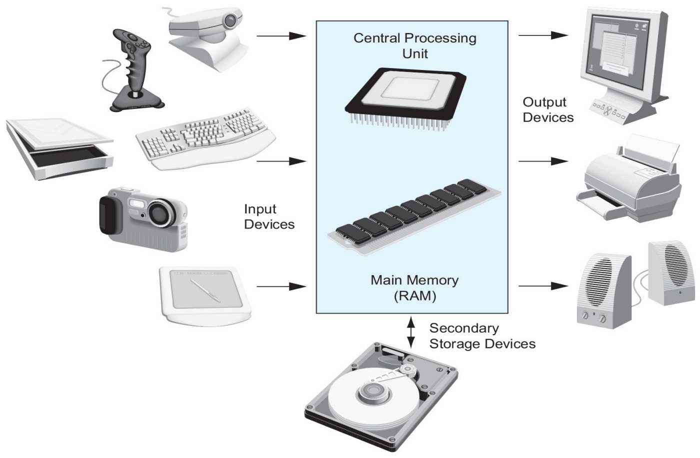


Figure 1.2. Components of a typical computer system.

Source: https://www.tutorialsmate.com/2020/04/computer-fundamentals-tutorial.html.

Let us take a deeper look at each of these elements individually.

1.2.1. The CPU

When a computer is engaged in the duties that a program has instructed it to execute, we refer to this as the computer running or executing the program in question. The central processing unit, sometimes known as the CPU, is the portion of a computer that is responsible for running programs. The CPU (central processing unit) is the most significant component in a computer since it is responsible for running software on the computer (Henning, 2000).

Computer processing units (CPUs) were massive devices constructed of mechanical and electrical components like vacuum tubes and switches in the early days of computing. Figure 1.3 depicts an example of such a gadget. In this picture we can see the ENIAC computer, which dates to the 1940s, being used by the two women seen. When the ENIAC was created in 1945, it was used to calculate weaponry ballistic tables for the United States Army. It is widely regarded as the world's first programmable electronic computer by many. This machine, which was essentially comprised of a single large CPU, stood 8 feet tall, measured 100 feet in length, and weighed 30 tons (Zhu et al., 2021).

CPUs are little chips that are referred to as microprocessors. A photographic depiction of a lab technician carrying a new microprocessor is seen in Figure 1.4. Microprocessors, along with being significantly smaller than the old-fashioned electromechanical CPUs found in initial computers, are also significantly more effective.

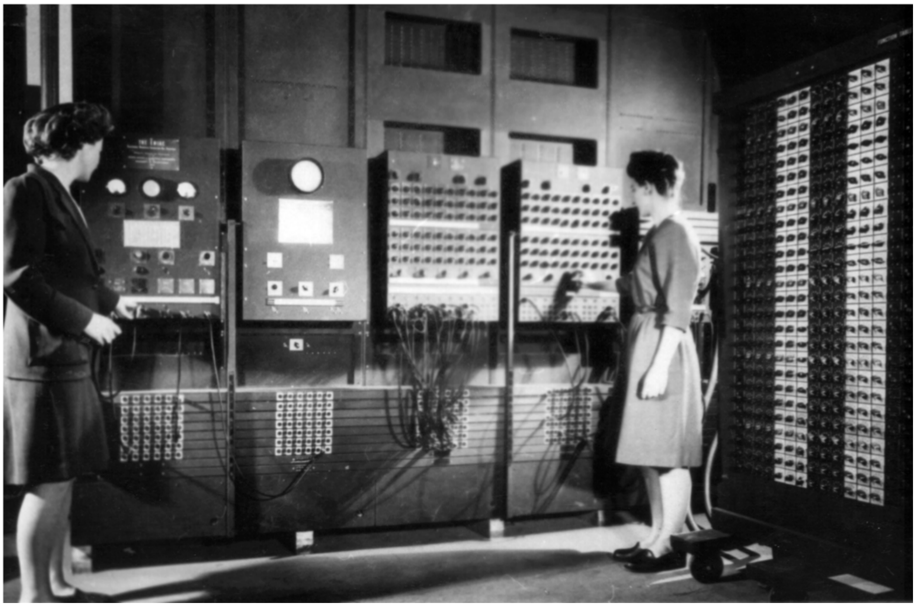


Figure 1.3. The ENIAC computer.

Source: https://www.indiatimes.com/technology/news/eniac-75-years-old-world-1st-programmable-digital-computer-534387.html.

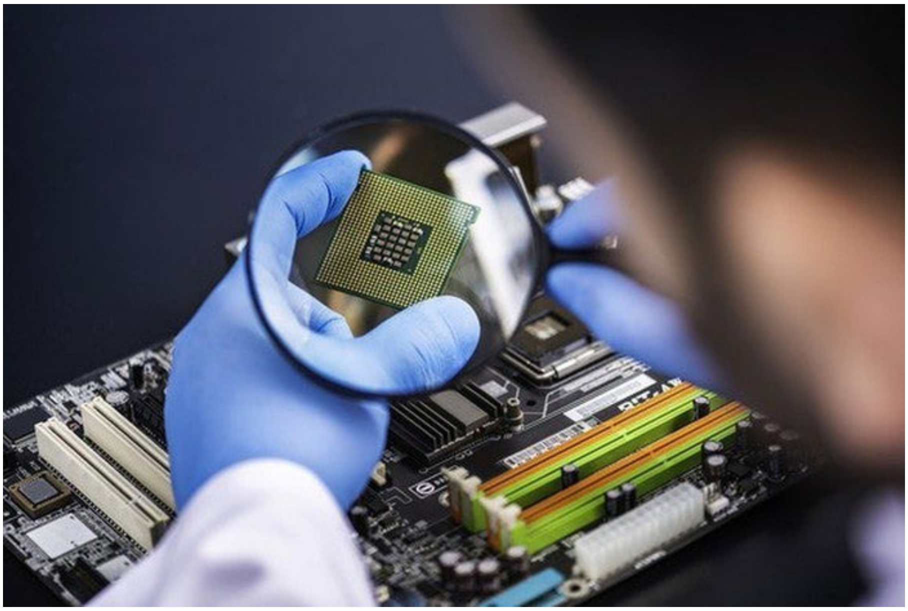


Figure 1.4. A lab technician holds a modern microprocessor.

Source: https://www.fool.com/investing/2021/01/15/why-intels-competitive-edge-is-crumbling/.

1.2.2. Main Memory

Consider main memory to be the computer's work area. This is where the computer keeps a program and the data that the program is working with while it is executing. Let us say you are writing an essay for one of your classes and you are using a word processing tool. Both the word processing program and the essay are saved in the main memory while you do this (Abali et al., 2021).

RAM, or Random-access memory, is the term for main memory. The CPU can instantly gain access to data stored in any arbitrary position in RAM, hence the name. RAM is a sort of unstable memory that is only employed for short-term storage as a program is executing. The contents in RAM are removed when the computer is shut off. RAM is stored in chips inside your computer, such as the ones depicted in Figure 1.5.

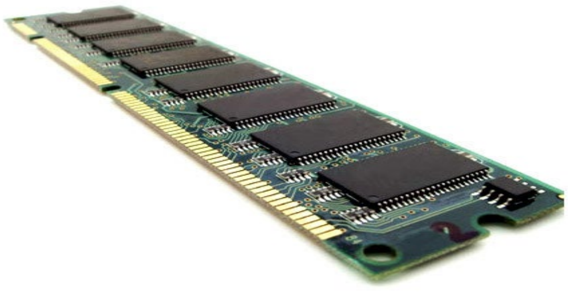


Figure 1.5. Memory chips (photo courtesy of IBM corporation).

Source: https://www.indiamart.com/proddetail/ram-memory-chip-2686040191.html.

1.2.3. Secondary Storage Devices

Secondary storage is a sort of memory that can keep data for a long time even if the computer is turned off. Normally, programs are stored in secondary memory and transferred into main memory only when needed. Vital data is also stored in secondary storage, like word processing documents, salary data, and inventory records (Babad et al., 1976).

The disc drive is the most popular form of secondary storage device. A disc drive saves data by imprinting it magnetically onto a circular disc. A disc drive is usually installed inside the casing of most computers. External disc drives are also accessible, which link to one of the computer's communication ports. External hard drives can be employed to make backup copies of vital files or to transfer data from one computer to another (Summer, 1967).

Aside from external disc drives, a variety of devices have been developed for copying and transporting data between computers. Floppy disc drives have been popular for a long time. A floppy disc drive saves information on a tiny floppy disc that may be removed. Floppy discs, on the other hand, have several drawbacks. They can only store a limited amount of data, are sluggish to obtain data, and are potentially untrustworthy. Recently, the usage of floppy disc drives has decreased considerably in support of more advanced devices such USB drives. USB drives are small devices that connect to a computer's USB interface and be seen as a disc drive to the operating system (OS). These drives, on the other hand, do not contain a disc. They keep data in flash memory, which is a unique sort of memory. USB drives often called flash drives or memory sticks, are affordable, dependable, and small enough to fit in your pouch (Babad et al., 1976).

For data storage, optical media such as DVD and CD are common. Data is encrypted as a sequence of pits on the disc surface rather than being recorded magnetically. A laser is used in DVD and CD drives to identify the pits and hence read the encoded data. Visual discs can carry a lot of data, and since recordable DVD and CD players are now popular, they are a handy way to make backup copies of your data (Chismar and Kriebel, 1982).

1.2.4. Input Devices

Input refers to any information that a computer receives from people or from additional devices. An input device is an element that takes data and provides it to a computer and is defined as follows: The keyboard, scanner, mouse, digital camera, and microphone are all examples of common input devices. Additionally, optical drives and disc drives can both be thought input devices since programs and data are regained from and encumbered into the computer's memory through them (Radwin et al., 1990).

1.2.5. Output Devices

Any data that a computer generates for people or for other devices is referred to as output. It might be anything from a sales report to a catalog of names to a graphic image. The data is transferred to an output device, which prepares and describes it in a visually appealing manner. Video screens and printers are two common types of output devices. Since the system transfers data to disc drives and CD recorders so that it may be saved, they can also be termed output devices (Burdea et al., 1996).

1.3. SOFTWARE

Software is required for a computer to function properly. Everything that a computer performs, from the moment the power switch is turned on up to the moment the system is shut off, is controlled by software. Application software and system software are the two broad categories of software that may be found in most computer systems. Most computer programs may be classified into one of these two types (Bazeley, 2006).

1.3.1. System Software

System software is a term that refers to the programs that control and manage a computer's fundamental activities. The following categories of applications are commonly found in system software:

- **Operating Systems (OSs):** On a computer, an OS is a basic set of applications. The OS maintains all the computer's connected components, allows it to be stored to and accessed from storage devices, and permits other programs to function on the computer. Figure 1.6 depicts four prominent OSs: Windows Vista, Windows XP, Linux, and Mac OS X (Yan et al., 2010).

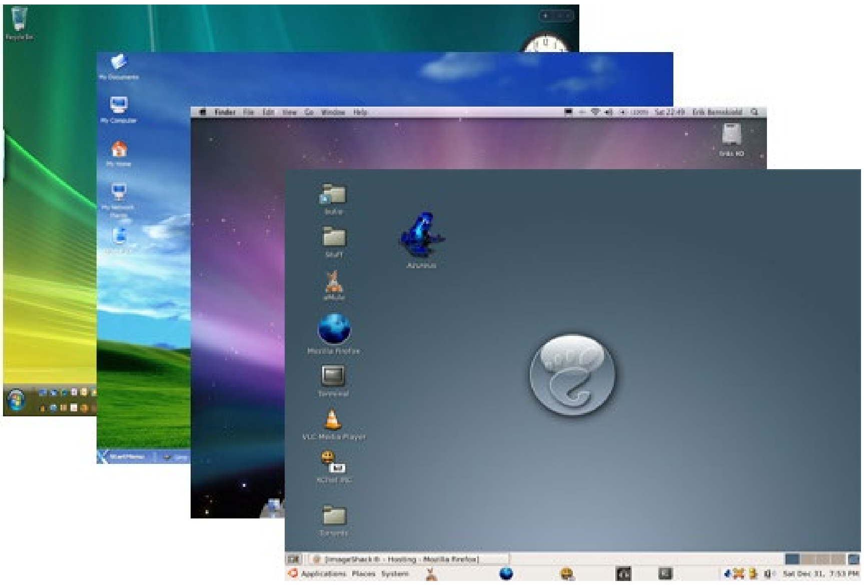


Figure 1.6. Screens from the Fedora Linux operating systems, Mac OS X, and Windows Vista.

Source: https://www.dmxzone.com/go/16325/os-smackdown-linux-vs-mac-os-x-vs-win-vista-vs-win-xp/.

- **Utility Programs:** A utility program accomplishes a specific duty that improves the computer's performance or protects data. Virus scanners, file compression programs, and data backup programs are examples of utility programs.
- **Software Development Tools:** The applications that programmers employ to create, edit, and test software are known as software development tools. Programs that belong under this category include assemblers, compilers, and interpreters.

1.3.2. Application Software

Application software refers to programs that make a computer helpful for daily tasks. These are the programs that most people use to consume most of their time on their computers. Figure 1.1 depicts screens from two regularly employed applications: Adobe Photoshop, a word processing program, and Microsoft Word, an image editing program, as shown at the start of this chapter. Spreadsheet programs, email programs, web browsers, and game programs are all examples of application software (Aerts et al., 2004).

1.4. HOW DO COMPUTERS STORE DATA?

All data stored in a computer is transformed to 0s and 1s sequences. The memory of a computer is divided into bytes, which are little storage units. A single byte of memory is just adequate to store a small integer or a single letter of the alphabet. A computer needs a lot of bytes to do anything useful. Most today's computers have millions, if not billions, of bits of memory.

Individually byte is distributed into eight bits, which are smaller storage spaces. The word "bit" refers to a binary digit. Bits are typically thought of by computer scientists as small switches that may be turned on or off. Bits, on the other hand, are not "switches" in the traditional sense. Bits are microscopic electrical elements that can store a negative or positive charge in most computer systems. A positive charge is thought of as a switch that is turned on, and a negative charge is thought of as a switch that is turned off by computer scientists. Figure 1.7 depicts how a computer engineer might conceptualize a bit of memory: as a set of switches, each of which can be flicked to the off or on state (Lehmann and Deutsch, 1995).

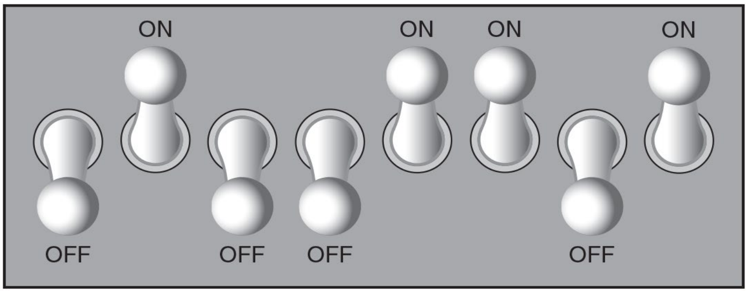


Figure 1.7. Byte as eight switches.

Source: https://www.pearsonhighered.com/assets/samplechapter/0/3/2/1/0321537114.pdf.

When a byte of data is gathered, the computer turns the eight bits of the byte into an off/on display that contains all the data. For instance, in Figure 1.8, the pattern on the left depicts how the numeral 77 would be kept in a byte, whereas the design on the right depicts how the letter A would be collected in a byte. We will go through how these shapes are created in more detail below.

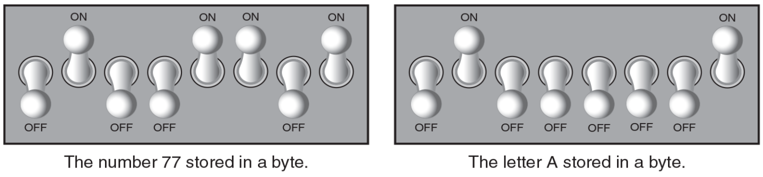

ON
ON
ON
ON
OFF
OFF
OFF
OFF
The number 77 stored in a byte.
ON
ON
OFF
OFF
OFF
OFF
OFF
OFF
The letter A stored in a byte.

Figure 1.8. The number 77 and the letter 'A' have different bit patterns.

Source: https://www.pearsonhighered.com/assets/samplechapter/0/3/2/1/0321537114.pdf.

1.4.1. Storing Numbers

A bit can only be employed to represent numbers in a very reduced number of situations. Because of the way bits work, they can represent one of two different values depending on whether they are turned on or off. Bits are used to represent numbers in computer systems. A bit that is turned off indicates the number 0, and a bit that is turned on indicates the number 1 in computer systems. This is a wonderful match for the binary numbering system, as you can see. In the binary numbering system, all numeric numbers are represented as a series of 0s and 1s. This is known as the binary representation of numbers. A number written in binary format is shown below as an illustration (Amit et al., 1985):

10011101

Every digit in a binary number has a value associated with it based on its place in the number. As illustrated in Figure 1.9, the position values are as follows: 2^0, 2^1, 2^2, 2^3, and so on, starting with the rightmost digit and working your way left. Figure 1.10 depicts the same diagram as in Figure 1.9, but with the position values determined. The position values are as follows: 1, 2, 4, 8, and so on, starting with the rightmost digit and working your way left (Grinko et al., 1995).

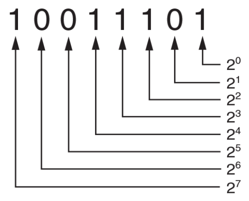


Figure 1.9. Binary digit values are expressed as powers of two.

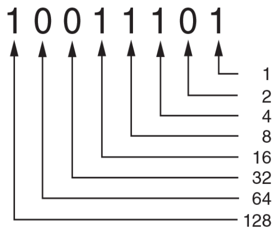


Figure 1.10. Binary digits and their values.

To find the amount of a binary number, just add all the 1s' position values. The position values of the 1s in the binary number 10011101, for example, are 1, 4, 8, 16, and 128. Figure 1.11 depicts this. Each of these position values adds up to 157. As a result, the binary number 10011101 has a value of 157.

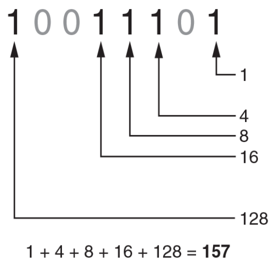


Figure 1.11. Determining the value of 10011101.

Figure 1.12 depicts how the number 157 is stored in a byte of memory. A bit in the on position represents each 1, and a bit in the off position represents each 0.

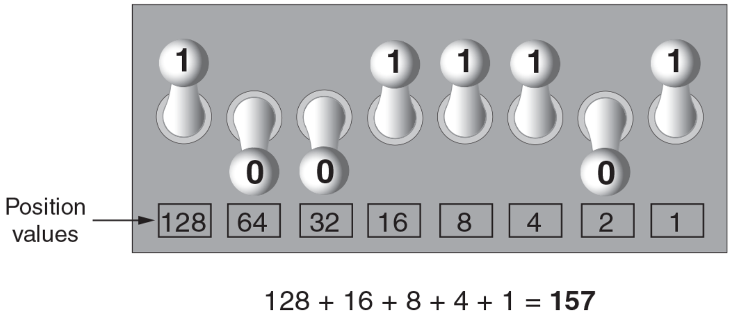


Figure 1.12. The bit pattern for 157.

What happens if you need to save a number that is greater than 255 characters? The solution is straightforward: utilize more than one byte. Consider the following scenario: we want to combine two bytes. This gives us a total of 16 bits. The standing values of those 16 bits would be 2^0, 2^1, 2^2, 2^3, and so on, all the way up to 2^{15}, depending on the bit position. As illustrated in Figure 1.13, the largest value that may be collected in two bytes is 65,535, which is the maximum possible value. If you need to hold a number that is larger than this, you will need to allocate more bytes (Okabe et al., 1984).

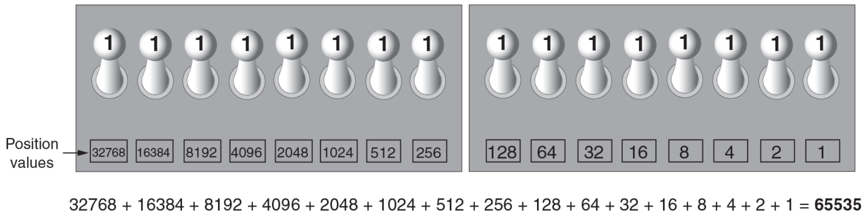


Figure 1.13. A huge number is represented by two bytes.

1.4.2. Storing Characters

A character is transformed to a numeric code before being stored in memory. The numerical code is then stored as a binary number in memory.

To display characters in computer memory, several coding methods have been created over time. The ASCII has historically been the most important of these coding schemes. The ASCII character set is made up of 128 numeric codes that signify punctuation marks, English letters, and other symbols. The ASCII code for the capital letter A, for example, is 65. The number 65 is stored in memory when you input a capital A on your computer keyboard. Figure 1.14 depicts this (Boonkrong and Somboonpattanakit, 2016).

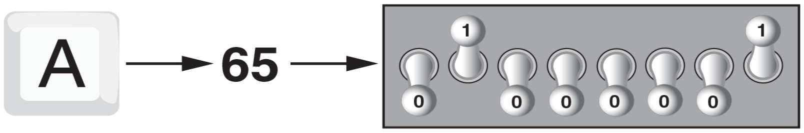


Figure 1.14. The number 65 is associated with the letter A in memory.

If you are curious, uppercase B has the ASCII code 66; uppercase C has the ASCII value 67, and so on. All the ASCII codes and the characters they represent are listed in Appendix C.

In the early 1960s, the ASCII character set was created, and it was subsequently accepted by nearly every computer manufacturer. However, ASCII is limiting since it only defines codes for 128 characters. In the early 1990s, the Unicode character set was created to address this issue. Unicode is a large encoding method that is comparable with ASCII but also capable of representing characters in a wide range of languages. Unicode is rapidly grown to be the de facto traditional character set in the computer industry (Melot and Tarascon, 2013).

1.4.3. Advanced Number Storage

You learned about numerals and how they were stored in memory previously. Maybe it appeared to you while reading that the binary numbering system can only be employed to express integer numbers starting with 0. The simple binary numbering approach we examined cannot express negative integers or real values (such as 3.14159) (Wang et al., 2015).

Negative and real numbers can be stored in memory by computers, but they must use encoding systems in addition to the binary numbering system to do so. Two's complement is used to encode negative integers, and floating-point notation is used to encode real numbers. You do not require to understand how these encoding methods operate; all you need to know is that they are employed to convert negative and real integers to binary (Greenhalgh et al., 1997).

1.4.4. Other Types of Data

The term "digital device" is frequently employed to explain computers. Whatever works with binary numbers is referred to as digital. A digital device is any device that operates with binary data, and digital data is anything that is collected in binary. We have looked at how characters and numbers are stored in binary in this part; however, computers can also operate with a variety of additional digital data.

Consider the photos you capture with your digital camera, for example. Pixels are small colored specks that make up these pictures. An image element is referred to as a pixel. Every pixel in a picture is transformed to a numeric code that describes the pixel's color, as illustrated in Figure 1.15. The numeric code is stored as a binary number in memory (Logan et al., 2012).

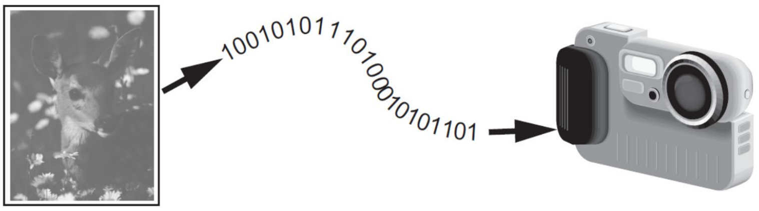


Figure 1.15. The binary format is used to store digital images.

The music that you listen to in iTunes, on your CD player, iPod, or MP3 device is also digitally encoded in some way. Samples are little bits of a digital song that are divided up into smaller portions. It is possible to store each sample in memory because each sample is transformed into a binary number. When a song is broken down into samples, the more closely it resembles the original music when it is played again. Approximately 44,000 samples per second are used to create CD-quality music (Iudici and Faccio, 2014).

1.5. HOW A PROGRAM WORKS?

The central processing unit (CPU) of a computer can just read guidelines that are expressed in machine language. Since it is extremely challenging for individuals to design full programs in machine language, additional programming languages have been developed to alleviate this difficulty.

We previously explained that the central processing unit is a highly significant part of a computer since it is the element of the computer that is responsible for running programs. The central processing unit (CPU) is sometimes referred to as the "computer's brain," and it is regarded as "clever." Even though these are typical analogies, it is important to recognize that the CPU is not a mind, and it is not intelligent. The central processing unit (CPU) is an electrical device that is intended to perform certain tasks. The CPU is specifically intended to conduct tasks such as the ones listed below (Faccio et al., 1979):

- Taking a chunk of data from the main memory and reading it;
- Adding two values;
- Taking one number and subtracting it from another;
- Multiplying two numbers;
- Multiplying one integer by another;
- Transferring information from one memory place to another;
- Trying to figure out if one number is the same as another.

The CPU makes simple actions on data. The CPU, on the other hand, accomplishes nothing on its own. It must be instructed what to do, and that is what a program is for. A program is nothing more than a set of guidelines that tell the CPU what to do (Card et al., 2018).

Every instruction in a program is an order to the CPU to carry out a certain task. Here is an example of a command that may be found in a program:

10110000

This is just a sequence of 1s and 0s to you and me. This, on the other hand, is a command to conduct an operation on a CPU. Since CPUs just understand machine language instructions, it is written in 0s and 1s. Machine language guidelines always have an inherent binary structure.

For any operation that a CPU is capable of, machine language instruction exists. For instance, there are instructions for adding integers deducting one number from another. The instruction set of a CPU refers to the whole set of instructions that it can perform (Slavin, 2008).

The machine language command that was previously displayed is only one of several. However, for the computer to accomplish anything useful, it needs a lot more than one instruction. Since these operations that a CPU can execute are so simple, a meaningful job can only be completed if the CPU makes many of them. If you need your computer to estimate the extent of interest you will earn this year from your savings account, for example, the CPU will have to execute a significant number of guidelines in the correct order. Thousands, if not millions, of machine language instructions, can be found in a single program (Olson, 2004).

A secondary storage device, like a disc drive, is usually employed to store programs. When you install software on your computer, the executable file is typically downloaded from a website to your computer's hard drive.

A program can be kept on a secondary storage device like a disc drive, but it must be transferred into RAM, or main memory, every time the processor performs it. Let us say you have a word processing application on your computer's hard drive. To run the software, double-click the program's icon with your mouse. The software is transferred from the disc into the main memory because of this. The CPU of the machine then runs the main memory copy of the application. This procedure is depicted in Figure 1.16 (Davenport, 1999).

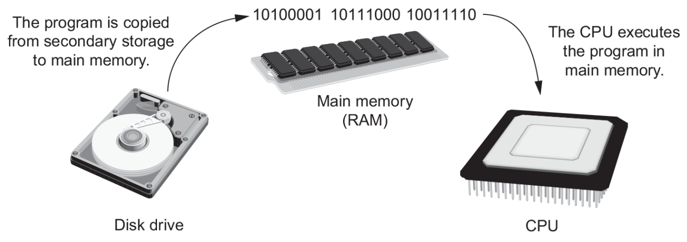

The program is copied from secondary storage to main memory.
10100001 10111000 10011110
The CPU executes the program in main memory.
Main memory (RAM)
Disk drive
CPU

Figure 1.16. After copying a program into the main memory, it is run.

The fetch-decode-execute cycle is the process that a CPU goes across as it executes the instructions of a program. This cycle, which is made up of three phases, is frequent for each program instruction. The steps are (Eckert, 1987):

- **Fetch:** A program is a set of instructions written in machine language. The next instruction is read from memory into the CPU in the first phase of the cycle.
- **Decode:** It is a binary number that signifies the command to the computer's central processing unit (CPU) to conduct a certain task. When the CPU decodes an instruction that has just been retrieved from memory, it may identify which operation it should do.
- **Get Data and Execute:** The operation is performed, or executed, as the final stage in the cycle. These processes are depicted in Figure 1.17.

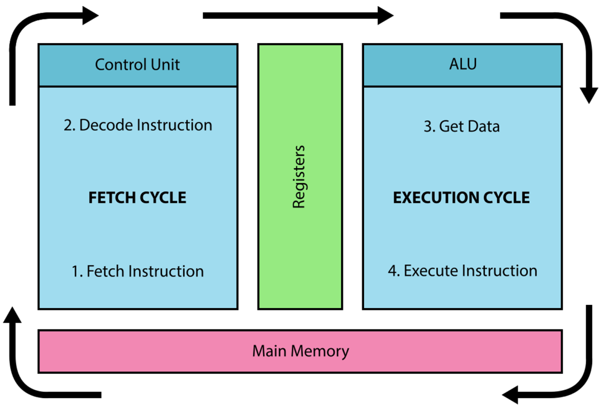


Figure 1.17. The cycle of fetch-decode-execute.

Source: https://www.pinterest.com/pin/438115869999300657/.

1.5.1. From Machine Language to Assembly Language

Only programs written in machine language can be executed by computers. A program can include dozens or even millions of binary instructions, as previously said, and developing such a program would be extremely laborious and time-consuming. It would also be difficult to program in machine language since inserting a 0 or 1 in the wrong location will result in an error (Ahmed et al., 2010).

While a computer's CPU can just comprehend machine language, writing programs in that language is impracticable. As a result, assembly language was created as an alternative to machine language in the early days of computing. Assembly language employs mnemonics, which are short phrases that replace binary digits in instructions. In assemblage language, for example, the mnemonic add usually means to add numbers, Mul usually means to multiply numbers, and mov usually means to transfer a value to a memory address. When writing a program in assembly language, a programmer can utilize short mnemonics instead of binary integers (Graham and Ingerman, 1965).

The CPU, on the other hand, cannot run assembly language programs. Because the CPU can only read machine language, an assembly program is needed to convert an assembly language program to machine code. Figure 1.18 depicts this procedure. The CPU may then run the machine language program that the assembler has built (Feldman, 1979).

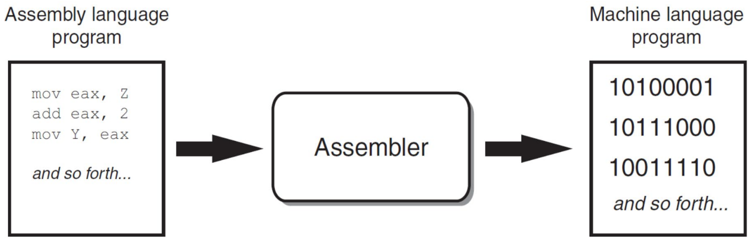


Figure 1.18. An assembler converts a program written in assembly language into a machine language program.

Source: https://www.educba.com/assembly-language-vs-machine-language/.

1.5.2. High-Level Languages

Assembly language eliminates the need for binary machine language guidelines, but it is still not devoid of its drawbacks. Assembly language is essentially a straight replacement for a machine language and it necessitates a thorough understanding of the CPU. Even the simplest program in assembly language necessitates the writing of a huge number of instructions. Assembly language is referred to as a low-level language (Halang and Stoyenko, 1990).

The 1950s saw the emergence of a new generation of programming languages known as high-level languages. A high-level language enables you to write powerful and complicated programs without having to understand how the CPU works or write a huge number of low-level instructions. Furthermore, most high-level languages employ simple terms. For instance, in COBOL, a programmer might write the following command to show the message. On the computer screen, hello world!

DISPLAY “Hello, world.”

Python is a high-level programming language that will be used throughout this book. The message Hello world would be shown in Python with the following instruction: ‘Hello world!’ print (Kennedy et al., 2004).

In assembly language, doing the same thing would take multiple instructions and a thorough understanding of how the CPU interacts with the computer’s output device is necessary to have when writing an assembly language program. As this instance shows, high-level languages permit programmers to focus on the goals they need their programs to do rather than the intricacies of how the CPU will execute such programs.

Thousands of high-level languages have been developed since the 1950s. Several of the best languages are included in Table 1.1.

Table 1.1. Programming Languages (Classen et al., 2011)

Language	**Description**
Ada	Ada was developed in the 1970s mainly for use by the United States Department of Defense. Countess Ada Lovelace, a significant and important person in the world of computers, is honored by the language’s name.
BASIC	All-purpose for beginners Symbolic Instruction Code is a speech known that was created in the early 1960s with the goal of being easy to learn for novices. There are many multiple variations of BASIC available today.

FORTRAN	The first high-level programming language was TRANslator. It was created in the 1950s to handle complicated mathematical operations.
COBOL	The common commercial-oriented language (CBOL) was developed in the 1950s for business applications.
Pascal	Pascal was established in 1970 with the intention of being used to teach programming. Blaise Pascal, a mathematician, physicist, and philosopher, was honored with the language's name.
C and C++	Bell Laboratories produced the strong general-purpose languages C and C++ (pronounced "c plus plus"). The C and C++ programming languages were established in 1972 and 1983, respectively.
C#	The letter "c sharp" is pronounced as "c sharp." This programming language was developed by Microsoft in the year 2000 for the purpose of developing applications that run on the Microsoft.NET framework.
Java	Sun Microsystems developed Java in the early 1990s and released it to the public. It may be employed to create applications that run on a single computer or that operate across the Internet via a web server, among other things.
JavaScript	JavaScript, which was developed in the 1990s, is a scripting language that may be employed in online pages. Even though its name, JavaScript is not linked to the Java programming language.
Python	Python, the programming language that we will be using in this book, is a general-objective programming language that was developed in the early 1990s. It has gained popularity in both corporate and educational applications in recent years.
Ruby	Ruby is a common-purpose programming language that was developed in the 1990s. It is based on the C programming language. It is becoming increasingly popular as a programming language for applications that operate on web servers.
Visual Basic	Visual basic is a software development environment developed and programming language by Microsoft that enables programmers to construct Windows-based programs in a short period of time. The first version of VB was developed in the early 1990s.

1.5.3. Key Words, Operators, and Syntax: An Overview

Each high-level language has its specific set of specified terms that should be used by the programmer while writing a program. Keywords or reserved words are the terms that make up a high-level programming language. Each keyword has a distinct meaning and cannot be utilized for anything else. You saw an example of a Python statement that prints a message on the screen using the keyword print previously. Many of the Python important words are shown in Table 1.2 (Rutherford, 1999).

Table 1.2. The Python Keywords

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

Programming languages feature operators that perform numerous actions on data in addition to keywords. All programming languages, for example, contain math operators that do arithmetic. The + sign is an operator that adds an additional integer in Python and most other languages. The following adds 12 and 75 to the total:

75 + 12

There are many more hands in the Python language, most of which you will understand as you read this book. Aside from important terms and operators, every language has its particular syntax, which is a collection of guidelines that should be observed to the letter while constructing a program. Syntax rules specify how important words, operators, and punctuation characters should be employed in a program. When studying a programming language, it is necessary to master the grammar rules for that language. Statements are the specific instructions used to build a program in a high-level programming language. A programming statement can be made up of punctuation, operators, keywords, and other programming components that are organized in the correct order to complete a task (Ertl and Gregg, 2003).

1.5.4. Compilers and Interpreters

Programs created in a high-level language must be translated language since the CPU only understands machine language instructions. The programmer will use either a compiler or an interpreter to translate a program depending on the language it was written in (Tanenbaum et al., 1983).

A compiler is a program that converts a program written in a high-level language into a machine language program. After that, the machine language program may be run whenever it is required (Figure 1.19). Compiling and executing are two distinct operations, as depicted in the diagram.

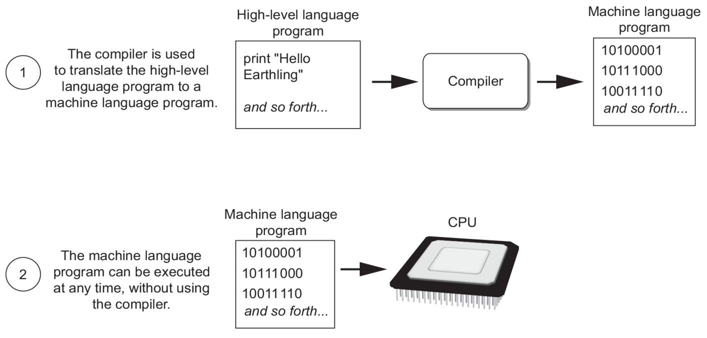

1
The compiler is used to translate the high-level language program to a machine language program.
High-level language program
print "Hello Earthling"
and so forth...
Compiler
Machine language program
10100001
10111000
10011110
and so forth...
2
The machine language program can be executed at any time, without using the compiler.
Machine language program
10100001
10111000
10011110
and so forth...
CPU

Figure 1.19. Compiling and running a high-level application.

Source: https://tutorials.one/computer-science-engineering/.

The interpreter in the Python language is a software that both interprets and implements the guidelines in a high-level language program. Each individual instruction in the program is read by the interpreter, which transforms it to machine language guidelines and then performs them instantly. This procedure is repeated for each of the program's instructions. Figure 1.20 illustrates this method. Interpreters seldom generate separate machine language programs since they mix translation and execution (Danvy, 2008).

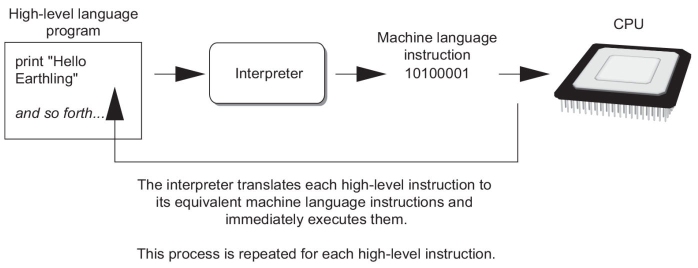

High-level language program
print "Hello Earthling"
and so forth...
Interpreter
Machine language instruction
10100001
CPU
The interpreter translates each high-level instruction to its equivalent machine language instructions and immediately executes them.
This process is repeated for each high-level instruction.

Figure 1.20. Using an interpreter to run a high-level program.

Source: https://slideplayer.com/slide/7417033/.

Source code refers to the statements that a programmer gives in a high-level language. Typically, a programmer writes the code for a program into a text editor and then saves it to the computer's disc. The programmer then uses an interpreter or a compiler to convert the code into a machine language program that can be executed. However, if the code has a syntactic issue, it will not be translated. A syntax error occurs when a crucial word is misspelled, a punctuation character is missing, or an operator is used incorrectly. When this occurs, the interpreter or compiler generates an error message identifying a syntactic mistake in the program. The programmer fixes the problem and then tries to translate the program again (Rossum, 2007).

1.6. USING PYTHON

The Python translator can run Python programs saved in files or perform Python statements input at the keyboard interactively. IDLE, an integrated development environment for Python, makes the process of developing, running, and testing programs easier (Sanner, 1999).

1.6.1. Installing Python

Before you can test any of the programs in this book or develop your own programs, you must first ensure that Python is installed and configured correctly on your machine. If you work in a computer lab, this has most likely already been done. If you have your personal computer, you may install Python from the included CD by following the instructions in Appendix A (Rossum and Boer, 1991).

1.6.2. The Python Interpreter

Python is an interpreted language. The Python interpreter is one of the components installed when you connect the Python language to your computer. The Python interpreter is software that reads and executes Python programming commands (Uieda et al., 2010).

The interpreter has two modes of operation: interactive and script. The interpreter pauses for you to write Python statements on the keyboard in interactive mode. The interpreter executes a statement after you write it and then waits for you to type another. The interpreter examines the contents of the file containing Python statements in script mode. A Python program or a Python script is the name for such a file. As it reads the Python program, the interpreter performs each statement (Mészárosová, 2015).

1.6.3. Interactive Mode

Once Python is installed and configured on your machine, you may start the interpreter in collaborative mode by running the following control at the OS's command prompt:

Python

If you are using Windows, you may also go to the Start menu and choose All Programs. You should notice a software group called Python 2.5 or something like that. There should be a Python item in this program group (command line). This menu option launches the Python interpreter in interactive mode when you click it (Frydenberg and Xu, 2019).

When you start the Python interpreter in interactive mode, you will see something like this in the console window:

```
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit
(Intel)] on win32
Type "help," "copyright," "credits" or "license" for more information.
>>>
```

The >>> you see is a prompt from the interpreter, indicating that it is waiting for you to input a Python statement. Let's give it a go. A print statement, which enables a message to be shown on the screen, is one of the most basic statements you may make in Python. The following sentence, for example, causes the notice to appear. Python programming is entertaining! to be exhibited:

```
print 'Python programming is fun!'
```

It's worth noting that we've written Python programming is enjoyable following the word print. Between a pair of single-quote marks, quotation marks are required, but they will not be used.

Shown. They merely indicate the start and finish of the text we want to show. Here's how you'd enter this print statement at the interpreter's prompt:

```
>>> print 'Python programming is fun!'
```

When you press the Enter key after inputting the sentence, the Python interpreter runs it, as illustrated above:

```
>>> print 'Python programming is fun!' [ENTER] Python programming is
fun!
```

```
>>>
```

The >>> prompt occurs after the message has been shown, indicating that the translator is waiting for you to enter another statement. Let's look at another scenario. We've entered two print statements in the following example session.

```
>>> print 'To be or not to be' [ENTER]
To be or not to be
>>> print 'That is the question.' [ENTER] That is the question.
>>>
```

The interpreter will display a message if you input a sentence improperly in interactive mode. This will help you learn Python by allowing you to use interactive mode. You may test out new sections of the Python language in interactive mode and get instant feedback from the interpreter as you learn them.

On a Windows machine, press Ctrl-Z subsequently Enter to exit the Python interpreter in interactive mode. Ctrl-D on a Mac, Linux, or UNIX computer (Chaudhury et al., 2010).

1.6.4. Writing Python Programs and Running Them in Script Mode

The statements you type in interactive mode are not preserved as a program, even though they are valuable for testing code. They are simply carried out, and the results are shown on the screen. You store a series of Python statements in a file if you wish to save them as a program. Then you utilize the Python translator in script mode to run the application (Newville, 2011).

Let's say you want to develop a Python program that shows the three lines of text below:

```
Nudge nudge
Wink wink
Know what I mean?
```

To write the program, create a file with the following statements using a simple text editor such as Notepad:

```
print 'Nudge nudge' print 'Wink wink' print 'Know what I mean?'
```

When you save a Python program, you give it a name that ends in.py, indicating that it is a Python program. For instance, you may save the previously demonstrated program as test.py. To launch the application, navigate to the list where the file is saved and enter the subsequent command from the OS command prompt (Price and Barnes, 2015):

Python test.py

This switches the Python translator to script mode and affects the statements in test.py to be executed. The Python interpreter terminates after the program is completed.

1.6.5. The IDLE Programming Environment

The preceding sections explained how to use the OS command line to launch the Python interpreter in interactive or script mode (Figure 1.21) (Puckette, 1991).

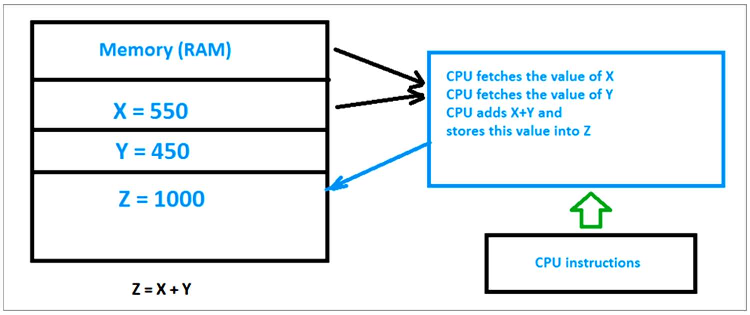
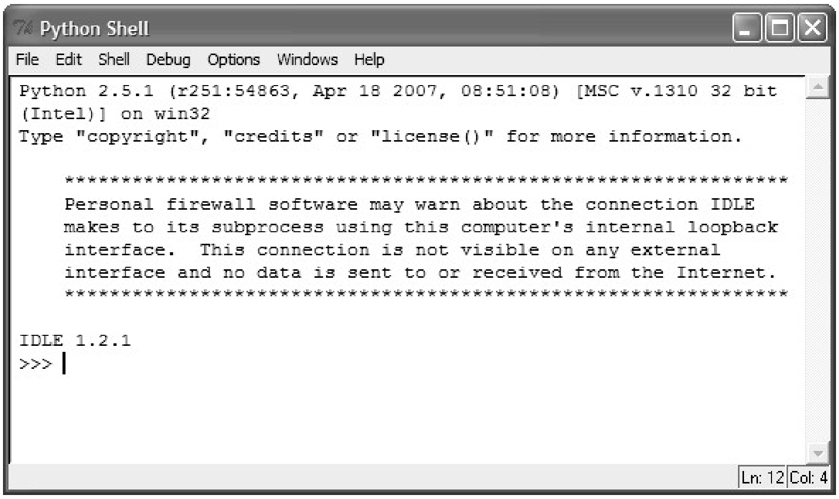


Figure 1.21. A typical computer program.

Source: https://www.softwaretestinghelp.com/basics-of-computer-programming/.

During the installation of the Python programming language, an application entitled IDLE, which is named after the Python programming language, will be automatically installed. IDL (Integrated Development Environment) is an acronym that holds for Integrated Development Environment. When you start IDLE, the window seen in Figure 1.22. You'll see that the >>> timely displays in the IDLE window, which indicates that the translator is now operating in collaborative mode. Python statements can be typed into this prompt, and the results will be shown in the IDLE window (Swinehart et al., 1986).

IDLE also has a developed-in text editor that includes features that are specially designed to assist you in the development of Python applications. In the IDLE editor, for example, code may be "colorized" so that key phrases and other sections of a program are shown in different hues. This contributes to making programs easier to understand. Create programs, save, and run them with IDLE’s program-writing environment. A brief introduction to IDLE is provided in Appendix B, which also guides you all through the process of writing, saving, and running a Python program (Stroud et al., 1988).



```
Python 2.5.1 (r251:54863, Apr 18 2007, 08:51:08) [MSC v.1310 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.2.1
>>> 
```

Figure 1.22. IDLE programming.

Source: https://web.mit.edu/6.s189/www/handouts/GettingStarted.html.

REFERENCES

1. Abali, B., Shen, X., Franke, H., Poff, D. E., & Smith, T. B., (2001). Hardware compressed main memory: Operating system support and performance evaluation. *IEEE Transactions on Computers, 50*(11), 1219–1233.
2. Aerts, A. T. M., Goossenaerts, J. B., Hammer, D. K., & Wortmann, J. C., (2004). Architectures in context: On the evolution of business, application software, and ICT platform architectures. *Information & Management, 41*(6), 781–794.
3. Ahmed, A., Appel, A. W., Richards, C. D., Swadi, K. N., Tan, G., & Wang, D. C., (2010). Semantic foundations for typed assembly languages. *ACM Transactions on Programming Languages and Systems (TOPLAS), 32*(3), 1–67.
4. Amit, D. J., Gutfreund, H., & Sompolinsky, H., (1985). Storing infinite numbers of patterns in a spin-glass model of neural networks. *Physical Review Letters, 55*(14), 1530.
5. Aroca, R. V., Caurin, G., & Carlos-SP-Brasil, S., (2009). A real time operating systems (RTOS) comparison. In: *WSO-Workshop de Sistemas Operacionais* (Vol. 12, pp. 1–10).
6. Ashraf, M. U., Fouz, F., & Eassa, F. A., (2016). Empirical analysis of HPC using different programming models. *International Journal of Modern Education & Computer Science, 8*(6), 3–12.
7. Atkinson, M. P., & Buneman, O. P., (1987). Types and persistence in database programming languages. *ACM Computing Surveys (CSUR), 19*(2), 105.
8. Babad, J. M., Balachandran, V., & Stohr, E. A., (1976). Management of program storage in computers. *Management Science, 23*(4), 380–390.
9. Balsamo, S., Personè, V. D. N., & Inverardi, P., (2003). A review on queueing network models with finite capacity queues for software architectures performance prediction. *Performance Evaluation, 51*(2–4), 269–288.
10. Bazeley, P., (2006). The contribution of computer software to integrating qualitative and quantitative data and analyses. *Research in the Schools, 13*(1), 64–74.
11. Ben-Akiva, M., De Palma, A., & Isam, K., (1991). Dynamic network models and driver information systems. *Transportation Research Part A: General, 25*(5), 251–266.

12. Boonkrong, S., & Somboonpattanakit, C., (2016). Dynamic salt generation and placement for secure password storing. *IAENG International Journal of Computer Science, 43*(1), 27–36.
13. Botha, C. P., (2006). *Technical Report: DeVIDE—The Delft Visualization and Image Processing Development Environment, 31*, 1–49.
14. Burdea, G., Richard, P., & Coiffet, P., (1996). Multimodal virtual reality: Input-output devices, system integration, and human factors. *International Journal of Human-Computer Interaction, 8*(1), 5–24.
15. Burgess, C. J., & Saidi, M., (1996). The automatic generation of test cases for optimizing Fortran compilers. *Information and Software Technology, 38*(2), 111–119.
16. Card, D., Kluve, J., & Weber, A., (2018). What works? A meta-analysis of recent active labor market program evaluations. *Journal of the European Economic Association, 16*(3), 894–931.
17. Chaudhury, S., Lyskov, S., & Gray, J. J., (2010). PyRosetta: A script-based interface for implementing molecular modeling algorithms using Rosetta. *Bioinformatics, 26*(5), 689–691.
18. Chen, J. B., Endo, Y., Chan, K., Mazieres, D., Dias, A., Seltzer, M., & Smith, M. D., (1995). The measured performance of personal computer operating systems. *ACM SIGOPS Operating Systems Review, 29*(5), 299–313.
19. Chen, J. B., Endo, Y., Chan, K., Mazieres, D., Dias, A., Seltzer, M., & Smith, M. D., (1996). The measured performance of personal computer operating systems. *ACM Transactions on Computer Systems (TOCS), 14*(1), 3–40.
20. Chismar, W., & Kriebel, C. H., (1982). Notes II comment on modeling the productivity of computer systems. *Management Science (pre-1986), 28*(4), 446.
21. Classen, A., Boucher, Q., & Heymans, P., (2011). A text-based approach to feature modeling: Syntax and semantics of TVL. *Science of Computer Programming, 76*(12), 1130–1143.
22. Crookes, D., Benkrid, K., Bouridane, A., Alotaibi, K., & Benkrid, A., (2000). Design and implementation of a high-level programming environment for FPGA-based image processing. *IEE Proceedings-Vision, Image, and Signal Processing, 147*(4), 377–384.

23. Danvy, O., (2008). Defunctionalized interpreters for programming languages. *ACM SIGPLAN Notices, 43*(9), 131–142.
24. Davenport, T. E., (1999). The federal clean lakes program works. *Water Science and Technology, 39*(3), 149–156.
25. Denning, P. J., & Buzen, J. P., (1978). The operational analysis of queueing network models. *ACM Computing Surveys (CSUR), 10*(3), 225–261.
26. Eckert, R. R., (1987). Kicking off a course in computer organization and assembly/machine language programming. *ACM SIGCSE Bulletin, 19*(4), 2–9.
27. Ertl, M. A., & Gregg, D., (2003). The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism, 5*, 1–25.
28. Faccio, W. J., (1979). Stimulating and rewarding invention: How the IBM awards program works. *Research Management, 22*(4), 24–27.
29. Feldman, J. A., (1979). High level programming for distributed computing. *Communications of the ACM, 22*(6), 353–368.
30. Feurzeig, W., Papert, S., Bloom, M., Grant, R., & Solomon, C., (1970). Programming-languages as a conceptual framework for teaching mathematics. *ACM SIGCUE Outlook, 4*(2), 13–17.
31. Frydenberg, M., & Xu, J., (2019). Easy as py: A first course in python with a taste of data analytics. *Information Systems Education Journal, 17*(4), 4.
32. Gilmore, D. J., & Green, T. R. G., (1984). Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies, 21*(1), 31–48.
33. Graham, G. S., (1978). Guest editor's overview… queuing network models of computer system performance. *ACM Computing Surveys (CSUR), 10*(3), 219–224.
34. Graham, M. L., & Ingerman, P. Z., (1965). An assembly language for reprogramming. *Communications of the ACM, 8*(12), 769–772.
35. Greenhalgh, T., (1997). How to read a paper: Statistics for the non-statistician. I: Different types of data need different statistical tests. *BMJ, 315*(7104), 364–366.
36. Grinko, I., Geerts, A., & Wisse, E., (1995). Experimental biliary fibrosis correlates with increased numbers of fat-storing and Kupffer cells, and portal endotoxemia. *Journal of Hepatology, 23*(4), 449–458.

37. Halang, W. A., & Stoyenko, A. D., (1990). Comparative evaluation of high-level real-time programming languages. *Real-Time Systems, 2*(4), 365–382.
38. Heller, J., & Logemann, G. W., (1966). PL/I: A programming language for humanities research. *Computers and the Humanities, 2*, 19–27.
39. Henning, J. L., (2000). SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer, 33*(7), 28–35.
40. Horn, M. S., Solovey, E. T., Crouser, R. J., & Jacob, R. J., (2009). Comparing the use of tangible and graphical programming languages for informal science education. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vol. 3, No. 1, pp. 975–984).
41. Iudici, A., & Faccio, E., (2014). What program works with bullying in school setting? Personal, social, and clinical implications of traditional and innovative intervention programs. *Procedia-Social and Behavioral Sciences, 116*, 4425–4429.
42. Kennedy, K., Koelbel, C., & Schreiber, R., (2004). Defining and measuring the productivity of programming languages. *The International Journal of High Performance Computing Applications, 18*(4), 441–448.
43. Knuth, D. E., & Pardo, L. T., (1980). The early development of programming languages. *A History of Computing in the Twentieth Century, 2*, 197–273.
44. Konstantinides, K., & Rasure, J. R., (1994). The Khoros software development environment for image and signal processing. *IEEE Transactions on Image Processing, 3*(3), 243–252.
45. Le Sage, T., Bindel, A., Conway, P. P., Justham, L. M., Slawson, S. E., & West, A. A., (2011). Embedded programming and real-time signal processing of swimming strokes. *Sports Engineering, 14*(1), 1–14.
46. Lehmann, E. D., & Deutsch, T., (1995). Application of computers in diabetes care a review. I. Computers for data collection and interpretation. *Medical Informatics, 20*(4), 281–302.
47. Liu, H., (2020). Design and application of micro course in fundamentals of computers. *International Journal of Emerging Technologies in Learning (iJET), 15*(11), 17–28.

48. Logan, B. E., (2012). Essential data and techniques for conducting microbial fuel cell and other types of bioelectrochemical system experiments. *ChemSusChem, 5*(6), 988–994.

49. Martinovic, G., Balen, J., & Cukic, B., (2012). Performance evaluation of recent Windows operating systems. *J. Univers. Comput. Sci., 18*(2), 218–263.

50. Melot, B. C., & Tarascon, J. M., (2013). Design and preparation of materials for advanced electrochemical storage. *Accounts of Chemical Research, 46*(5), 1226–1238.

51. Mészárosová, E., (2015). Is python an appropriate programming language for teaching programming in secondary schools? *International Journal of Information and Communication Technologies in Education, 4*(2), 5–14.

52. Newville, M., (2001). IFEFFIT: Interactive XAFS analysis and FEFF fitting. *Journal of Synchrotron Radiation, 8*(2), 322–324.

53. Okabe, T., Yorifuji, H., Yamada, E., & Takaku, F., (1984). Isolation and characterization of vitamin-A-storing lung cells. *Experimental Cell Research, 154*(1), 125–135.

54. Olson, D. R., (2004). The triumph of hope over experience in the search for "what works": A response to slavin. *Educational Researcher, 33*(1), 24–26.

55. Ottenstein, K. J., Ballance, R. A., & MacCabe, A. B., (1990). The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In: *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (Vol. 2, No. 1, pp. 257–271).

56. Price, T. W., & Barnes, T., (2015). Comparing textual and block interfaces in a novice programming environment. In: *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (Vol. 2, No. 1, pp. 91–99).

57. Puckette, M., (1991). Combining event and signal processing in the MAX graphical programming environment. *Computer Music Journal, 15*(3), 68–77.

58. Radwin, R. G., Vanderheiden, G. C., & Lin, M. L., (1990). A method for evaluating head-controlled computer input devices using Fitts' law. *Human Factors, 32*(4), 423–438.

59. Rasure, J., Argiro, D., Sauer, T., & Williams, C., (1990). Visual language and software development environment for image processing. *International Journal of Imaging Systems and Technology, 2*(3), 183–199.
60. Rutherford, T. F., (1999). Applied general equilibrium modeling with MPSGE as a GAMS subsystem: An overview of the modeling framework and syntax. *Computational economics, 14*(1), 1–46.
61. Sage, D., & Unser, M., (2003). Teaching image-processing programming in java. *IEEE Signal Processing Magazine, 20*(6), 43–52.
62. Sanner, M. F., (1999). Python: A programming language for software integration and development. *J Mol Graph Model, 17*(1), 57–61.
63. Slavin, R. E., (2008). Perspectives on evidence-based research in education—What works? Issues in synthesizing educational program evaluations. *Educational Researcher, 37*(1), 5–14.
64. So, H. K. H., & Brodersen, R., (2008). A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Transactions on Embedded Computing Systems (TECS), 7*(2), 1–28.
65. Steere, D. C., Shor, M. H., Goel, A., Walpole, J., & Pu, C., (2000). Control and modeling issues in computer operating systems: Resource management for real-rate computer applications. In: *Proceedings of the 39th IEEE Conference on Decision and Control* (Vol. 3, pp. 2212–2221).
66. Stroud, C. E., Munoz, R. R., & Pierce, D. A., (1988). Behavioral model synthesis with cones. *IEEE Design & Test of Computers, 5*(3), 22–30.
67. Summer, C. E., (1967). Critique of: "An overview of management science and information systems." *Management Science, 13*(12), B-834.
68. Swift, J. A., & Mize, J. H., (1995). Out-of-control pattern recognition and analysis for quality control charts using lisp-based systems. *Computers & Industrial Engineering, 28*(1), 81–91.
69. Swinehart, D. C., Zellweger, P. T., Beach, R. J., & Hagmann, R. B., (1986). A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems (TOPLAS), 8*(4), 419–490.

70. Tanenbaum, A. S., Van, S. H., Keizer, E. G., & Stevenson, J. W., (1983). A practical tool kit for making portable compilers. *Communications of the ACM, 26*(9), 654–660.

71. Tanimoto, S. L., (1990). VIVA: A visual language for image processing. *Journal of Visual Languages & Computing, 1*(2), 127–139.

72. Tejada, J., Chudnovsky, E. M., Del Barco, E., Hernandez, J. M., & Spiller, T. P., (2001). Magnetic qubits as hardware for quantum computers. *Nanotechnology, 12*(2), 181.

73. Thomasian, A., & Bay, P. F., (1986). Analytic queueing network models for parallel processing of task systems. *IEEE Transactions on Computers, 35*(12), 1045–1054.

74. Ton, R. V. B. D. K., Mosterd, K. T. K. B., & Smeulders, A. W., (1994). Scillmage: A multi-layered environment for use and development of image processing software. *Experimental Environments for Computer Vision and Image Processing, 11*, 107.

75. Trichina, E., (1999). Didactic instructional tool for topics in computer science. *ACM SIGCSE Bulletin, 31*(3), 95–98.

76. Uieda, L., Ussami, N., & Braitenberg, C. F., (2010). Computation of the gravity gradient tensor due to topographic masses using tesseroids. *Eos Trans. AGU, 91*(26), 1–21.

77. Van, R. G., & De Boer, J., (1991). Interactively testing remote servers using the python programming language. *CWI Quarterly, 4*(4), 283–303.

78. Van, R. G., (2007). Python programming language. In: *USENIX Annual Technical Conference* (Vol. 41, No. 1, pp. 1–36).

79. Vandersteen, G., Wambacq, P., Rolain, Y., Dobrovolný, P., Donnay, S., Engels, M., & Bolsens, I., (2000). A methodology for efficient high-level dataflow simulation of mixed-signal front-ends of digital telecom transceivers. In: *Proceedings of the 37th Annual Design Automation Conference* (Vol. 4, pp. 440–445).

80. Walski, T. M., Brill, Jr. E. D., Gessler, J., Goulter, I. C., Jeppson, R. M., Lansey, K., & Ormsbee, L., (1987). Battle of the network models: Epilogue. *Journal of Water Resources Planning and Management, 113*(2), 191–203.

81. Wang, Y., Wei, H., Lu, Y., Wei, S., Wujcik, E. K., & Guo, Z., (2015). Multifunctional carbon nanostructures for advanced energy storage applications. *Nanomaterials, 5*(2), 755–777.

82. Wasserman, A. I., & Prenner, C. J., (1979). Toward a unified view of database management, programming languages, and operating systems—A tutorial. *Information Systems, 4*(2), 119–126.

83. Yan, K. K., Fang, G., Bhardwaj, N., Alexander, R. P., & Gerstein, M., (2010). Comparing genomes to computer operating systems in terms of the topology and evolution of their regulatory control networks. *Proceedings of the National Academy of Sciences, 107*(20), 9186–9191.

84. Zhu, J., Luo, A., Li, G., Zhang, B., Wang, Y., Shan, G., & Liu, L., (2021). Jintide: Utilizing low-cost reconfigurable external monitors to substantially enhance hardware security of large-scale CPU clusters. *IEEE Journal of Solid-State Circuits, 56*(8), 2585–2601.

CHAPTER 2

CLASSIFICATION OF COMPUTER PROGRAMS

CONTENTS

2.1. INTRODUCTION

A computer program is a set of instructions written in a programming language that a computer may perform or understand in imperative programming. A computer program is a collection of instructions in declarative programming.

Source code is the human-readable version of a computer program. As computers may only execute their native machine instructions, source code requires the execution of another computer program. As a result, utilizing the language's compiler, source code can be converted to machine instructions. (An assembler is used to convert machine language programs.) An executable is a name given to the generated file. Instead, source code can run in the interpreter of the language. The Java programming language generates an intermediate form that is subsequently processed by a Java interpreter (Wilson and Leslie, 2001).

If the operating system (OS) receives a request to run the executable, it loads it into memory and initiates a procedure to carry out the request (Silberschatz and Abraham, 1994). The central processing unit (CPU) would be switched to this procedure so that it may fetch and decode every machine instruction before executing them. As soon as the source code is required for implementation, the OS loads the relevant interpreter into memory and begins the execution of the procedure. The interpreter then puts the source code into memory, where it may be translated and executed one statement at a time by the processor (Tanenbaum and Andrew, 1990). Compared to launching an executable, running the source code is more time-consuming. In addition, the interpreter should be installed on the PC in question.

It is possible to make advances in the development of software as an outcome of advancements in computer hardware. Throughout the history of hardware, the work of computer programming has undergone significant transformations.

Charles Babbage had been motivated via Jacquard's loom to create the Analytical Engine in 1837 (McCartney and Scott, 1999). The names of the calculating device's elements had been taken from the textile industry. The yarn had been carried from the shop to be processed in the textile business. The gadget contained a "store," or memory, that could keep 1,000 numbers, each with 50 decimal digits (Tanenbaum and Andrew, 1990). Numbers were transported from the "storage" to the "mill" for processing. Two sets of perforated cards were used to program it. One set is for the operation's direction, while the other is for the input variables (McCartney and Scott, 1999; Bromley and Allan, 1998). Consequently, after spending over £17,000 of the British government's money, the thousands of cogged gears and wheels were never completely functional (Figure 2.1) (Tanenbaum and Andrew, 1990).

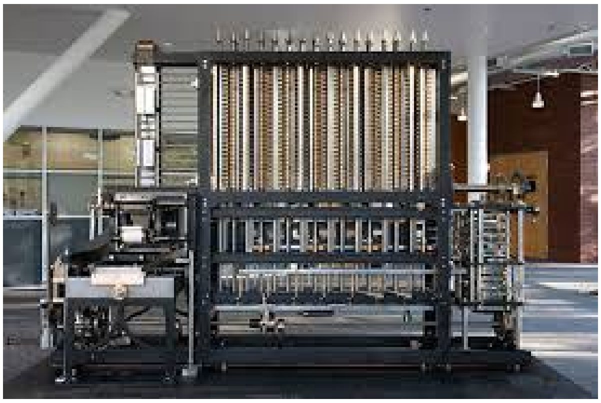


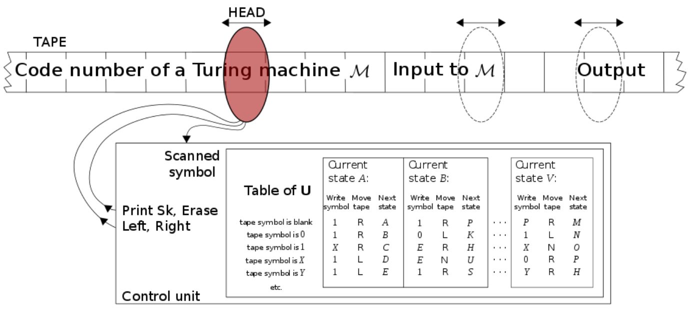
Figure 2.1. Analytical engine of Lovelace.

Source: https://www.csmonitor.com/Technology/2012/1210/Ada-Lovelace-What-did-the-first-computer-program-do.

Charles Babbage commissioned Ada Lovelace to write information on the Analytical Engine (1843) (Fuegi and Francis, 2003). The explanation included Note G, which described in detail how to use the Analytical Engine to compute Bernoulli numbers. Certain historians consider this note to be the world's first computer program (Tanenbaum and Andrew, 1990).

2.1.1. Universal Turing Machine

Alan Turing proposed the Universal Turing System in 1936, which is a theoretical device that may mimic any calculation that may be done on a Turing complete computer machine (Rosen and Kenneth, 1991). It has an endlessly long write or red tape and is a finite-state machine. While performing an algorithm, the machine may move the tape back and forth, altering its contents. The machine begins in the initial state, proceeds through a series of phases, and finally comes to a rest when it reaches the halt state (Figure 2.2) (Linz and Peter, 1990).


HEAD
TAPE
Code number of a Turing machine $\mathcal{M}$
Input to $\mathcal{M}$
Output
Scanned symbol
Print Sk, Erase Left, Right
Control unit
Table of U

	Current state A:			Current state B:				Current state V:		
	Write symbol	Move tape	Next state	Write symbol	Move tape	Next state		Write symbol	Move tape	Next state
tape symbol is blank	1	R	A	1	R	P	···	P	R	M
tape symbol is 0	1	R	B	0	L	K	···	1	L	N
tape symbol is 1	X	R	C	E	R	H	···	X	N	O
tape symbol is X	1	L	D	E	N	U	···	0	R	P
tape symbol is Y	1	L	E	1	R	S	···	Y	R	H
etc.										

Figure 2.2. Universal Turing machine.

Source: https://www.wikiwand.com/en/Universal_Turing_machine.

2.1.2. Relay-Based Computers

It was Konrad Zuse who created the Z3 computer in 1941, which was a programmable and digital computer. Zuse first became acquainted with the "Babbage Engine" in 1939, when seeking to file a German patent application for it. The Analytical Engine was in base-10, which made it simple to understand. Zuse realized that constructing a binary machine was a simple process. Telephone relays are 2-position switches that are either closed or open in nature (Stair and Ralph, 2003). The Z3 contained around 2,600 relays, with 1,800 dedicated to memory, 600 dedicated to arithmetic, and two hundred dedicated to the keyboard, punch tape reader, and display, among other things. The circuits enabled the creation of a floating-point computer with nine instructions. The Z3 was programmed using a customized keyboard and punch tape that was built specifically for it. Manual input had been accomplished using a calculator-style keyboard that supported decimal integers. The input was translated to binary by the machine, and the results were transmitted through a series of calculating modules. The result had been translated back to decimal and presented on a display panel at the bottom of the screen (Figure 2.3) (Weiss and Mark, 1994; Bach and Maurice, 1986).

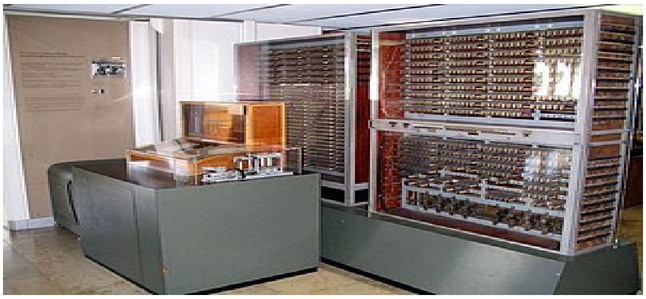


Figure 2.3. Zuse Z3 replica on display at Deutsches Museum in Munich.

Source: https://en.wikipedia.org/wiki/Computer_program.

Its successor, the Z4, had been created at the same time. (Z3 was destroyed by an airstrike on April 6, 1945). The Z4 was first produced in 1950 at the Federal Technical Institute in Zurich.

The Harvard Mark I was a programmable and digital computer created via IBM in 1944 (Stroustrup and Bjarne, 2013). The computer had 7 main units and supported twenty-three signed integer digits (Elgot et al., 1982):

- The machine's activities were directed by a single unit;
- One unit contained 60 dial switches for configuring the application constants;
- Multiplication and division were done with a single unit;
- One unit did addition and subtraction and stored the intermediate results in 72 registers;
- Interpolation was utilized to compute logarithmic functions with a single unit;
- Interpolation was utilized to compute trigonometric functions by using a single unit;
- The machine's output medium was either a typewriter printer or a punched card printer, and one unit was employed to direct it.

Harvard's Mark I was 3,304 relays and 530 miles of wire on my system. The input was given by two punched tape readers (Kernighan et al., 1988). The directions were typed in by one of the readers. Howard H. Aiken compiled a codebook that listed all of the known algorithms. A programmer punched the coded commands onto a tape from this book. The data to be processed was entered by the other reader.

Harvard's Mark IBM's 2 additional relay-based machines succeeded me (Koren, 2018):

- The Harvard Mark II, for example;
- The electronic selective sequence calculator (SSEC). Until August 1952, the SSEC was in operation.

2.1.3. ENIAC

Between July 1943 and the fall of 1945, the computer (ENIAC) and Electronic Numerical Integrator was created. It had been a Turing complete, general-purpose computer with circuits made out of 17,468 vacuum tubes (Haigh et al., 2016). It had been essentially a collection of Pascalines that had been linked together. Its 40 units weighed thirty tones, took up 1,800 square feet (167 m^2), and used $650 in power each hour (in 1940s money). There were 20 base-10 accumulators in it. It took up to 2 months to program the ENIAC. 3 function tables had to be moved to constant function panels since they had been on wheels (Kerrisk and Michael, 2010; Weik, 1961). Heavy black wires had been utilized to link function tables to function panels. Every function table had 728 knobs that rotated. Setting some of the 3,000 switches on the ENIAC was also part of the programming process. It took a week to debug a program. It operated at Aberdeen Proving Ground from 1947 to 1955, computing hydrogen bomb characteristics, forecasting weather patterns, and providing firing tables for artillery cannon aiming (Figure 2.4) (Jones et al., 2012).

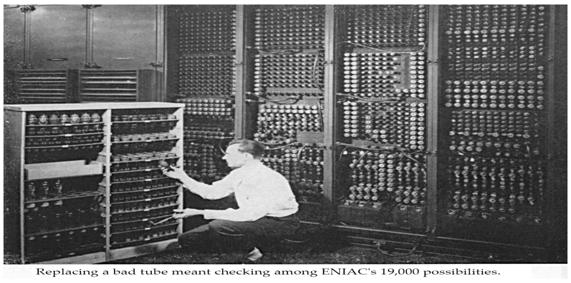

Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

Figure 2.4. Glenn A. Beck is changing a tube in ENIAC.

Source: https://commons.wikimedia.org/wiki/File:ENIAC-changing_a_tube.jpg.

2.1.4. Stored-Program Computers

A stored-program computer loads its commands into memory the same way it loads its data into memory instead of plugging in connections and flicking switches. As a consequence, the computer was able to be programmed rapidly and do computations at a high rate. The ENIAC was designed by Presper Eckert and John Mauchly. In a 3-page document dated February 1944, the 2 engineers introduced the stored-program notion. Dr John von Neumann started working on the project of ENIAC in September 1944 (Huang et al., 2017). Von Neumann released the first draft of the report on EDVAC on June 30 1945, that likened the computer's architecture with those of the human brain. Von Neumann architecture was the name given to the design. In 1949, the design had been utilized to build the EDSAC and EDVAC computers at the same time (McCartney and Scott, 1999). In 1961, the Burroughs B5000 was designed expressly for use with the Algol 60 programming language. The hardware included circuits to help with the compilation process (Tanenbaum and Andrew, 1990). The IBM System/360 had been a series of 6 computers released in 1964, each with the identical command set architecture. The Model thirty was the tiniest and most affordable. Customers might advance their applications while keeping the same software. The most expensive model was the Model 75. Multiprogramming was available on all System/360 models, allowing numerous processes to be stored in memory at the same time. Another process may compute while the first was waiting for input/output. Every model was supposed to be programmed in PL/1, according to IBM. COBOL, Fortran, and ALGOL programmers were assembled into a committee. The goal was to create a language that will replace Fortran and Cobol by being comprehensive, simple to utilize, and extendible. As a result, the language grew in size and complexity and it required a long time to build (Figure 2.5) (Wilson and Leslie, 2001).

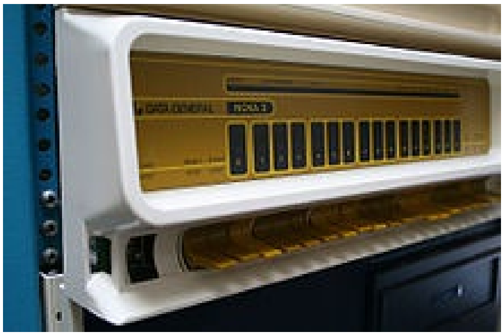


Figure 2.5. On a data general nova 3 from the mid-1970s, there are switches for manual input.

Source: https://en.wikipedia.org/wiki/Data_General_Nova.

Up to the 1970s, computers used front-panel switches for manual programming (Gordon and Michael, 1996) For reference, the computer program had been written on paper. A series of on/off settings had been utilized to indicate a command. An execution button was pushed when the setup was completed. After that, the procedure was repeated. Punched cards or paper tape was also used to automatically input computer programs. The beginning address was set through switches once the medium was loaded, and the execution button was hit (Schach and Stephen, 1990).

2.1.5. Very Large-Scale Integration

The VLSI circuit was a big breakthrough in software development (1964) (Tan et al., 2003). After World War II, bipolar junction transistors (the late 1950s) and point-contact transistors (1947) put on a circuit board superseded tube-based technology. In the 1960s, the aircraft sector adopted the integrated circuit chip (Figure 2.6) (Silberschatz and Abraham, 1994).

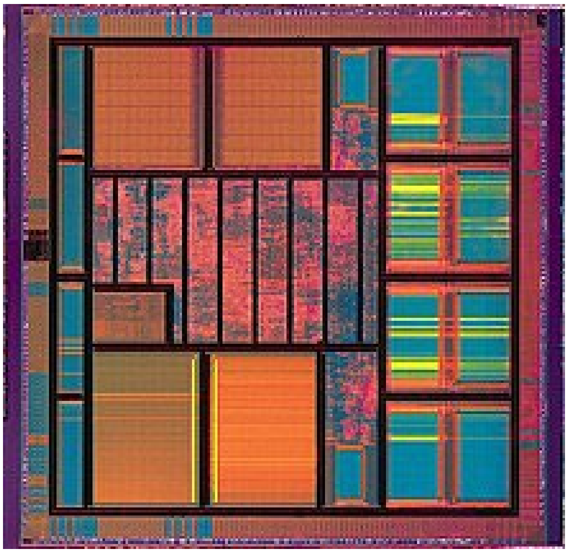


Figure 2.6. A VLSI integrated-circuit die.

Source: https://en.wikipedia.org/wiki/Very_Large_Scale_Integration.

Robert Noyce, a co-founder of Intel (1968) and Fairchild Semiconductor (1957), improved the field-effect transistor manufacturing technology (1963). The objective is to change a semiconductor junction's electrical resistance and conductivity. The Siemens technique has been the first one which transforms naturally existing silicate minerals into polysilicon rods (Lacamera and Daniele, 2018). The rods are subsequently transformed into a monocrystalline silicon boule crystal by the Czochralski technique. To make a capacitor, wafer substrate, the crystal is finely cut (Kernighan and Brian, 1984). The planar photolithography technique then integrates unipolar transistors, and resistors onto the wafer to create a matrix of MOS transistors. In integrated circuit chips, the MOS transistor is the fundamental component (Haviland and Keith, 1987).

Initially, the purpose of integrated circuit chips was determined during production. Controlling the flow of electricity shifted to programming a read-only memory (ROM) matrix in the 1960s (ROM). A 2-D arrangement of fuses resembled the matrix. The unnecessary connections were burned off during the embedding of instructions into the matrix (Tolpygo et al., 2016). Because there were several connections, firmware programmers created computer software to manage the burning on a separate chip. Programmable ROM was the name of the technology. The Intel 4,004 microprocessor was born in 1971 after Intel implemented the computer program on the chip (Figure 2.7).

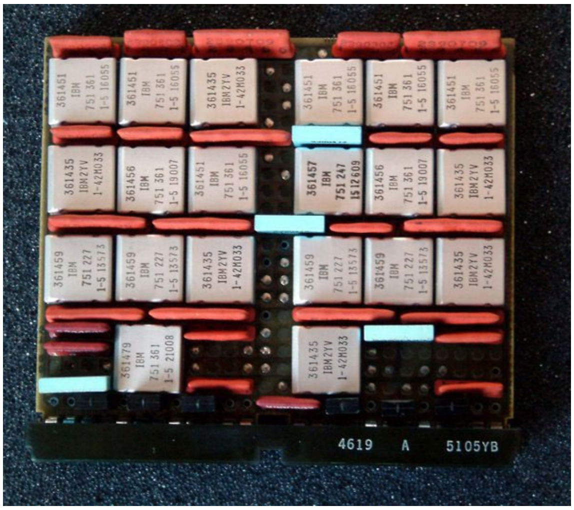


Figure 2.7. *IBM's system/360 (1964) CPU was not a microprocessor.*

Source: https://www.quora.com/Was-the-IBM-System-360-mainframe-computer-built-with-all-transistors-or-did-it-utilize-integrated-circuits.

The central processing unit (CPU) and *microprocessor* are now synonyms. But CPUs precede microprocessors. It used circuit boards with discrete components on ceramic substrates, for example as in the IBM System/360 (1964).

2.1.6. Sac State 8008

The Intel 4004 was a four-bit microprocessor that powered the Busicom calculator. Intel launched the Intel 8008, an eight-bit CPU, 5 months after it was first launched. The Sac State 8008 was the first^{t} microcomputer built with the Intel 8008 and directed by Bill Pentz (1972). It was created to store patient medical records. The computer had a disc OS that could operate a three-megabyte Memorex hard disc drive. It included a single console with a keyboard and color display (Figure 2.8) (Lee, 2000).

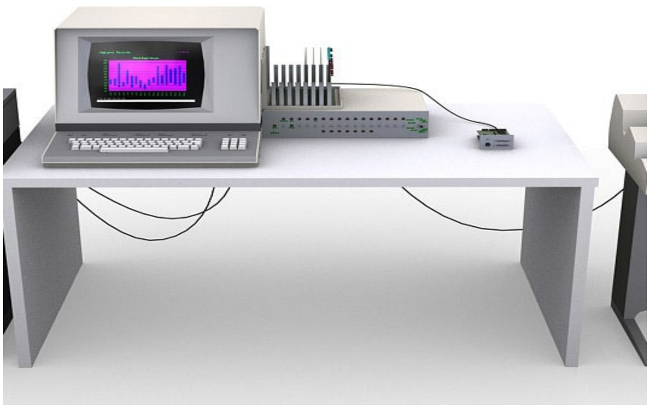


Figure 2.8. Sacramento State University's Intel 8008 microcomputer (1972) is shown by an artist.

Source: https://www.researchgate.net/figure/The-3D-reconstruction-of-the-Sac-State-8008-microcomputer-circa-1972-73-credit-Ryan_fig1_303697288.

IBM's basic assembly language (BAL) was used to program the disc OS. A BASIC interpreter was used to program the medical records application. The computer, on the other hand, had been an evolutionary dead-end due to its exorbitant cost. It was also designed for a particular purpose in a public university laboratory (Damer, 2011). Despite this, the effort aided in the creation of the Intel 8080 instruction set (1974).

2.1.7. x86 Series

When Intel updated the Intel 8080 to the Intel 8086 in 1978, the contemporary software development environment started. Intel modified the Intel 8086 to produce the Intel 8088 at a lower cost (Seiler et al., 2008). When IBM joined the personal computer market, they chose the Intel 8088 (1981) (Figure 2.9).

Figure 2.9. The original IBM personal computer (1981) utilized an Intel 8088 microprocessor.

Source: https://www.pcmag.com/news/project-chess-the-story-behind-the-original-ibm-pc.

Intel's microprocessor development accelerated as customer demand for personal computers (PC) grew. The x86 series refers to the development sequence. The x86 assembly language is a set of computer instructions that are backwards compatible. Machine commands stored in older microprocessors were carried over to newer microprocessors. Customers were allowed to buy the latest computers without needing to buy the latest application software as a result of this. The following are the primary types of instructions (Draper and Ingraham, 1968):

- Random-access memory instructions for setting and accessing integers and strings;
- Instructions for performing elementary arithmetic operations on integers using the integer arithmetic logic unit (ALU);
- Floating-point ALU commands for performing real-number arithmetic operations;
- Use call stack commands to allocate interface and memory with functions by pushing and popping words;

- SIMD (multiple data, single instruction) instructions boost performance when many processors are used to running a similar algorithm upon an arrangement of data.

2.1.8. Programming Environment

VLSI circuits allowed the programming environment to evolve from a computer terminal to a graphical user interface (GUI) computer (till the 1990s). Programmers were confined to a single shell operating in a command-line environment on computer terminals. The editing of Full-screen source code using a text-based user interface became feasible in the 1970s. The objective is to program in the language of programming, regardless of the technology accessible (Figure 2.10) (Zinnat, 2021).

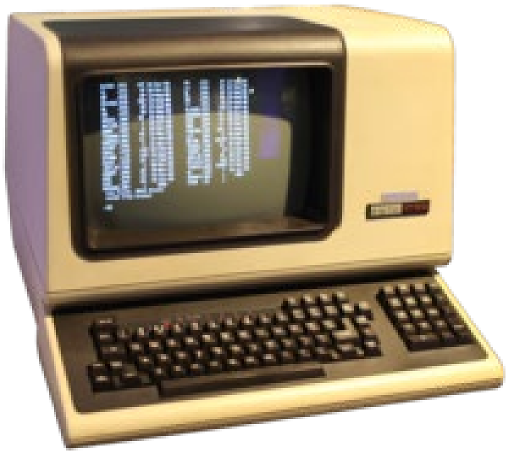


Figure 2.10. The DEC VT100 (1978) was an extensively utilized computer terminal.

Source: https://en.wikipedia.org/wiki/Computer_terminal.

2.2. SOFTWARE SYSTEMS

The properties of software systems, as well as their link to other aspects of software engineering environments, are discussed in further detail in the sections that follow. There are several classifications for software systems based on the way the tasks to be completed and the software system interact with one another (Nimmer et al., 1987).

2.2.1. Domain-Independent Software

The independence of the job to be executed and the class of jobs that may be executed are two characteristics of domain-based software (Zhang et al., 2003). The work is unaffected by the passage of time. Domain-agnostic software may be divided into two subtypes/LEH80/, /MABU87/:

- Specifiable systems (S-type); and
- Programmable systems (P-type).

In P-type systems, the criteria may be precisely described, and an implementation that precisely meets the criteria can be realized. The choosing of one of many good solutions is the procedure of development. Numeric and arranging procedures are instances of P-type systems.

A precise job specification may be supplied in the case of S-type systems, but only an estimated implementation is available. During the development phase, a solution must be identified that is as near to the original specifications as feasible. Game playing systems and various mathematics problems are instances of S-type systems. Chess algorithms are programmed with a specific goal in mind: "win each game." Moreover, we are aware that this program does not occur. All chess systems strive to meet the "Always win" condition as closely as possible (Atkins et al., 2011).

The definition, design, and implementation of a program may be done in that sequence to build domain-independent software. Before the design operations begin, the specification can be finished. Only comprehensive specifications are allowed using the specification techniques. Specification tools may verify if a specification is comprehensive.

In typical software development, domain-independent activities are uncommon. Usually, requirements alter or aren't understood. A software system's needs might vary on its own. The work to be completed is better understood after the establishment of a software system. New needs emerge, or the priority of existing requirements shifts. As a result, domain-dependent (DD) software systems emerge (Fleischmann, 1994).

2.2.2. Domain Dependent Software

The key feature of DD software is that its needs change over time. This occurs either because of the tasks to be accomplished change or as the system's presence has an impact on the real-world environment in a way /MABU87/. DD software systems are referred to as evolutionary systems in /MABU87/ and /LEH80/ (Hutson, 1997). According to /GlEDD84/ DD-systems are classified as either DD-software experimental (DDEX) or DD-software embedded (DDEM) systems. DDEX the development of the system is classified via an inherent ambiguity regarding the range of activities to be completed. Investigations into physical or economic phenomena are instances of this sort of software. DDEX software may lead to the design of software for a variety of sectors or applications, such as an economic model for usage in a management information system (Figure 2.11).

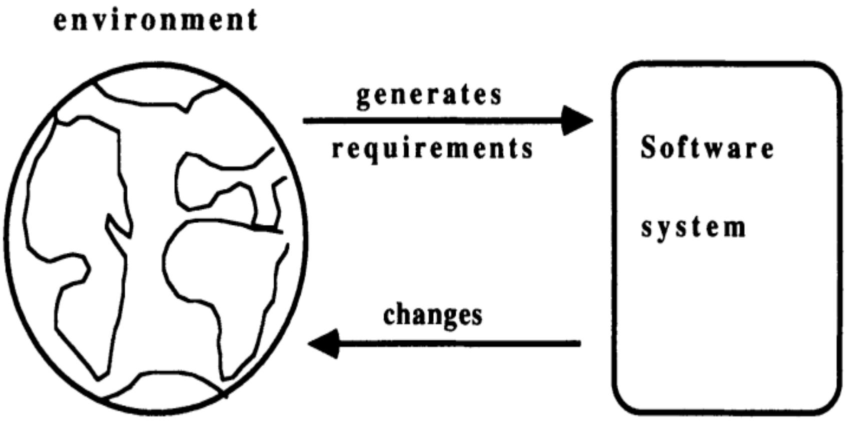


Figure 2.11. Domain-dependent software systems.

Source: https://link.springer.com/chapter/10.1007/978-3-642-78612-9_3?noAccess=true.

The connection between the class of jobs to be completed and the program is a feature of DDEM software. The program may modify the application area and, as a result, the expression of the work to be completed. Software engineering systems, office automation systems, consecutive generations of large-scale OS and factory control systems are instances of DDEM software. The diagram below depicts how a software system is dependent on its surroundings and how it alters its environment (Bar-Sinai et al., 2018).

Adequate life cycles and strategies should be used to construct domain-based software systems. Life cycles that begin with insufficient criteria and end with the design and execution of an unfinished system must be employed. The development procedure then moves on to a phase of requirement formulation. Then the design of the 2nd stage is prepared and implemented the 2nd stage, and so forth. This software development strategy also necessitates that the method of the specification allows for the declaration of partial criteria (Noldus, 1991). To survive the frequent changes, the design and design technique should be adaptable.

2.3. GENERAL BEHAVIOR OF SOFTWARE SYSTEMS

The software has two primary characteristics that may greatly enhance the complexity of a system (Riddle, 1979):

- Time limitations; and
- Requirements for fault tolerance.

Because these factors can impact the software engineering environment and methodologies, specific methods must be employed to build a system that takes these factors into account (Hayes-Roth et al., 1995).

2.3.1. Limitation of Time

The nature of the activity requires software systems to reply in a certain period. The following system types may be distinguished as they correspond to growing time requirements (Shen and Yu, 2018).

2.3.1.1. Requirements for Non-Real-Time

Batch systems are an instance of software systems with no time constraints. When a batch system completes a task, it takes the input, calculates the conclusions, and outputs the results.

2.3.1.2. Requirements for Weak Real-Time

Dialog systems are an instance of software systems with low time requirements. The environment is represented by humans in this illustration. Queries are entered in, and the system must respond promptly. Long reaction times are irritating for users, but they are seldom fatal (Piteira et al., 2013).

2.3.1.3. Hard Real-Time Requirements

A hard real-time system should adhere to strict time constraints to function properly. Procedure control systems and communication are instances of hard real-time systems (Moser et al., 1996).

2.3.2. Communication Systems

Many computers may be linked together to form a network that sends and receives data packets. A computer's environment is made up of all the other computers with which it is linked. By delivering information packets to other computers in the network, a computer may connect with its surroundings. Communication rules (also known as communication protocols) can require the receiver to send back an acknowledgement packet to certify that a data packet has reached its destination (Rullan, 1997). The sender of the data package considers that the receiver is not operating or that the packet was lost in the network if the return packet does not come within a specific amount of time. In communication protocols, time-outs are critical. This is the sole way to determine if a computer in a network is unavailable. If a computer does not respond within a certain amount of time, the sender thinks that the receiver is down. This instance demonstrates the significance of time restrictions in communication systems (Lee and Lee, 2004).

2.3.3. Process Control Systems

When a computer system controls a technical procedure, like an assembly line, the software in the computer intermingles with the technical system. As a result of this interaction, two significant real-time needs emerge:

- The control system must do certain actions at a specific moment; and
- Within the technical procedure, the control system must react to stochastically occurring occurrences.

The system should provide a specific reaction time for both needs; otherwise, the effects might be devastating (Luus, 1975).

A technique of specification should allow for the declaration of temporal constraints. Simulators may be used to get a preliminary sense of whether a system would work as expected. The specification approaches that are very successful when specifying editors, maybe entirely ineffective when specifying communication systems.

2.3.4. System Reliability

According to /RALETR78/, system dependability is connected to how well a system performs the required service. The fault intolerance technique and the fault tolerance technique are 2 ways to build highly dependable systems. Fault intolerance comprises all known strategies for ensuring that software contains no flaws, like requirements descriptions, proving, breakthroughs, and testing. Moreover, experience has shown that such procedures may only help to decrease problems and never ensure their removal. In software engineering, the fault intolerance approach is commonly employed (Kuo and Prasad, 2000). Fault tolerance strategies include approaches that provide an adequate service despite the presence of problems that remain after the usage of fault intolerant procedures. To replace broken components, redundant parts are inserted into a system. These items are not required to perform the stated service in the absence of problems. To be fault-tolerant, a software system must make the required design decisions and employ acceptable procedures. The necessity for fault tolerance while building a software system may greatly enhance its complexity. To build fault-tolerant systems, effective techniques for specifying the kind and degree of fault tolerance must be used. These techniques should be implemented in the software engineering environment (Martin-Löf, 1982).

2.4. PROGRAM TYPES

The application's type, the computer system to be utilized, and eventually the programmer's individual preference all influence whether a parallel or sequential program is designed. Because the methodologies for generating sequential and parallel programs are so dissimilar, a decision should be finalized before commencing program development, as well as the software engineering environment that is appropriate for the style of programming being employed (Turski et al., 1978).

2.4.1. Sequential Programs

Sequential programs are defined through a sequence of statements connected by a single control thread. A sequential program is run by a single procedure from the user's perspective; while there can be latent parallelisms like vector or array computing (that is not taken into account here). Only the output/input behavior is considered in sequential programs (Isard et al., 2007). The relationship between a program's starting and terminating states is known as input/output behavior. Sequential programming has been used to create a lot of commercial and technical software. Editors and compilers for PC are common instances.

2.4.2. Parallel or Concurrent Programs

Concurrent or parallel programs are made up of a series of statements linked via various control threads. The computer system and the criteria determine whether a sequential or concurrent program should be built (Gregory, 1987).

The hardware system or compilers usually hide fine-grain parallelism. The compiler divides a sequential program into multiple concurrent procedures. A multiprocessor system runs several procedures in parallel. Processor pipelines may run one statement whilst transferring the next statement from memory to the processor. Fine-grain parallelism has little impact on software engineering methodologies since it is masked by hardware design or compilers (Chandy et al., 1991). This is not to be confused with structural parallelism. In certain situations, structural parallelism is quite important. The communication software systems and process control, in particular, are modeled as systems of concurrent procedures are discussed in this chapter. Specific description and programming approaches must be employed to implement systems of this kind. This chapter's major focus is on programming approaches. In the following chapter, we go through these strategies in great depth (Hwang et al., 1984).

2.5. COMPUTER ARCHITECTURE

The CPUs and memory are the most important parts of a computer. The relationships among such components may be used to differentiate various kinds of computer systems (Sommer et al., 2013).

2.5.1. Centralized Computer Systems

There is just one CPU in such systems that have accessibility to the memory (shown in Figure 2.12) (Thota et al., 2018).

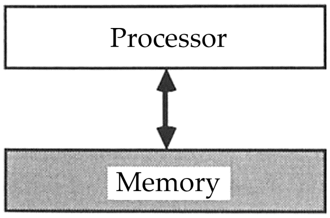


Figure 2.12. Structure of centralized computer system.

Source: https://link.springer.com/chapter/10.1007/978-3-642-78612-9_3?noAccess=true.

2.5.2. Multiprocessor Systems

A single memory is shared by multiple processors (shown in Figure 2.13) (Iqbal et al., 2010).

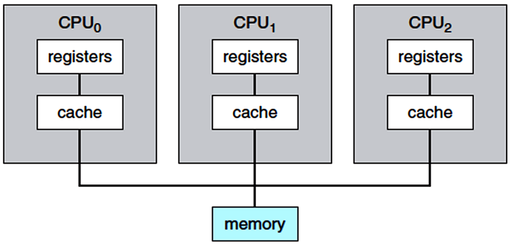


Figure 2.13. Structure of a multiprocessor system.

Source: https://zitoc.com/multiprocessor-system/.

2.5.3. Distributed Systems

A distributed system can be very complex to define precisely. Several disputed definitions may be discovered such as /NEHM88b/, /ENSL78/, / SLKR87/. In the following section, we use the following definition from /BSTA88/: Multiple independent processors, each with their primary memory, make up a physically dispersed system. Every communication in a distributed system is performed through messages that are exchanged over a messaging transportation system (Figure 2.14) (Agha, 1985).

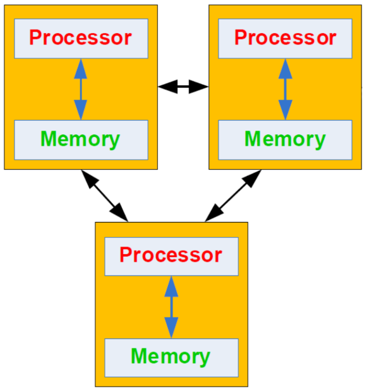


Figure 2.14. A distributed system's structure.

Source: https://www.researchgate.net/figure/Distributed-System-Structure_fig2_287975451.

2.5.4. Distributed Multiprocessor Systems

A message transit system connects numerous multiprocessor systems to form a distributed multiprocessor system (Figure 2.15) (Kuhl et al., 1980).

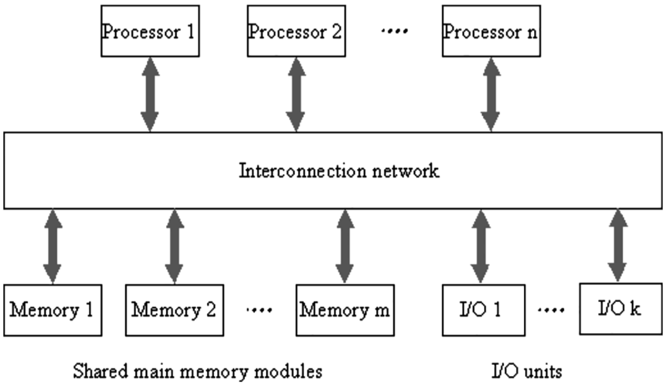


Figure 2.15. A distributed multiprocessor system's structure.

Source: https://edux.pjwstk.edu.pl/mat/264/lec/main119.html.

Tools like pre-compilers and compilers may hide the system architecture and the program kind. It is only necessary to note a system's unique design in software development if it is to be utilized explicitly. As previously said, developing a distributed system that works on distributed system architecture necessitates the use of unique methodologies (Agha, 1985). We presume that a developer is aware that he must construct a distributed software system. In this approach, he may make explicit usage of the benefits of distributed systems (Robins et al., 2003).

2.6. EXAMPLES

To demonstrate the categories introduced in the preceding sections, many software systems are studied. A compiler, an editor, a rapid Fourier transformation, a flight reservation system, chess software, a flight control system, and a physical model are all detailed in Table 2.1 (Hasselbring et al., 2006). Some characteristics may be included in the software, but they are not required. Optional qualities are highlighted in Table 2.1 by a question mark, for example, an editor may be fault-tolerant but does not have to be. A flight reservation system, for example, may function on a distributed or centralized system, including a distributed database system (Park et al., 1997; Liskov, 1972).

Table 2.1. Properties of Compiler, an Editor, a Chess Program, a Fast Fourier Transformation, a Physical Model, a Flight Reservation System, and a Flight Control System

Properties / Program	Relationship to the Environment				General Behavior				Program Type		System Architecture			
	Domain independent		Domain Dependent		fault	Time Constraints								
Program	S-Type	P-Type	Emb.	Exp.	tol.	no	weak	hard	seq.	par.	cent.	Mult.	dist.	dist. Mult.
Fasst Fourier	x					x			x		x			
physical model				x		x			x		x			
Chess		x				x			x		x			
Editor			x		x ?		x		x		x			
Compiler	x					x			x		x			
Flight Reser-vation			x				x			x	x		x	
Flight Control System			x		x			x		x		x	x	x

2.7. DISCUSSION

We recognize that the classifications presented in the preceding sections cannot be complete, but they do cover a broad range of software systems. Artificial intelligence (AI) is one topic that isn't fully addressed. AI necessitates knowledge, yet because the information is vast, difficult to identify precisely, and continually changing, it necessitates the adoption of unique ways to express it. It is feasible to address AI issues without applying AI techniques, such as theorem proving and natural language comprehension; likewise, such solutions are unlikely to be particularly successful. AI systems are now implemented in PROLOG or LISP. Such languages may be classified as sequential, but it doesn't tell us anything about the jobs they're supposed to execute. This demonstrates that our software taxonomy is merely 1st step toward expressing the complexity of a software system; yet, we believe it may be useful in conventional software fields.

REFERENCES

1. Agha, G. A., (1985). *Actors: A Model of Concurrent Computation in Distributed Systems* (Vol. 1, p. 1–10). Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab.
2. Atkins, D., Neshatian, K., & Zhang, M., (2011). A domain-independent genetic programming approach to automatic feature extraction for image classification. In: *2011 IEEE Congress of Evolutionary Computation (CEC)* (pp. 238–245). IEEE.
3. Bach, M. J., (1986). *The Design of the UNIX Operating System* (p. 152). Prentice-Hall, Inc. ISBN 0-13-201799-7.
4. Bar-Sinai, M., Weiss, G., & Shmuel, R., (2018). BPjs: An extensible, open infrastructure for behavioral programming research. In: *Proceedings of the 21st ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems: Companion Proceedings* (pp. 59–60).
5. Bromley, A. G., (1998). Charles Babbage's analytical engine, 1838 (PDF). *IEEE Annals of the History of Computing, 20*(4), 29–45. doi: 10.1109/85.728228. S2CID 2285332.
6. Chandy, K. M., & Kesselman, C., (1991). Parallel programming in 2001. *IEEE Software, 8*(6), 11–20.
7. Damer, B., (2011). TIMELINES the DigiBarn computer museum: A personal passion for personal computing. *Interactions, 18*(3), 72–74.
8. Draper, R. D., & Ingraham, L. L., (1968). A potentiometric study of the flavin semiquinone equilibrium. *Archives of Biochemistry and Biophysics, 125*(3), 802–808.
9. Elgot, C. C., & Robinson, A., (1982). Random-access stored-program machines, an approach to programming languages. In: *Selected Papers* (pp. 17–51). Springer, New York, NY.
10. Fleischmann, A., (1994). Classification of software system types. In: *Distributed Systems* (pp. 35–44). Springer, Berlin, Heidelberg.
11. Fuegi, J., & Francis, J., (2003). Lovelace & Babbage and the creation of the 1843 'notes.' *IEEE Annals of the History of Computing, 25*(4), 16–26.
12. Gordon, M., (1996). *From LCF to HOL: A Short History* (Vol. 5, No. 2, pp. 5–12).

13. Gregory, S., (1987). *Parallel Logic Programming in PARLOG: The Language and its Implementation* (Vol. 1, pp. 1–15). Addison-Wesley Longman Publishing Co., Inc.
14. Haigh, T., Priestley, P. M., Priestley, M., & Rope, C., (2016). *ENIAC in Action: Making and Remaking the Modern Computer* (Vol. 1, pp. 1–20). MIT press.
15. Hasselbring, W., & Reussner, R., (2006). Toward trustworthy software systems. *Computer, 39*(4), 91, 92.
16. Haviland, K., (1987). *Unix System Programming* (p. 121). Addison-Wesley Publishing Company. ISBN 0-201-12919-1.
17. Hayes-Roth, B., Pfleger, K., Lalanda, P., Morignot, P., & Balabanovic, M., (1995). A domain-specific software architecture for adaptive intelligent systems. *IEEE Transactions on Software Engineering, 21*(4), 288–301.
18. Hennessy, J. L., & Patterson, D. A., (2011). *Computer Architecture: A Quantitative Approach* (Vol. 1, p. 1–14). Elsevier.
19. Huang, J., Gharavi, H., Yan, H., & Xing, C. C., (2017). Network coding in relay-based device-to-device communications. *IEEE Network, 31*(4), 102–107.
20. Hutson, L. J., (1997). *A Representational Approach to Knowledge and Multiple Skill Levels for Broad Classes of Computer-Generated Forces* (Vol. 1, p. 1–13). Air Force Inst of Tech Wright-Patterson AFB OH.
21. Hwang, K., & Faye, A., (1984). *Computer Architecture and Parallel Processing, 1*, 1–17.
22. Iqbal, S. M. Z., Liang, Y., & Grahn, H., (2010). Parmibench-an open-source benchmark for embedded multiprocessor systems. *IEEE Computer Architecture Letters, 9*(2), 45–48.
23. Isard, M., Budiu, M., Yu, Y., Birrell, A., & Fetterly, D., (2007). Dryad: Distributed data-parallel programs from sequential building blocks. In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems* (pp. 59–72).
24. Kernighan, B. W., & Ritchie, D. M., (1988). *The C Programming Language* (2nd edn., p. 31). Prentice-Hall. ISBN 0-13-110362-8.
25. Kernighan, B. W., (1984). *The Unix Programming Environment* (p. 201). Prentice-Hall. ISBN 0-13-937699-2.
26. Kerrisk, M., (2010). *The Linux Programming Interface* (p. 121). No Starch Press. ISBN 978-1-59327-220-3.

27. Koren, I., (2018). *Computer Arithmetic Algorithms* (Vol. 3, No. 2, pp. 1–5). AK Peters/CRC Press.
28. Kuhl, J. G., & Reddy, S. M., (1980). Distributed fault tolerance for large multiprocessor systems. In: *Proceedings of the 7th Annual Symposium on Computer Architecture* (pp. 23–30).
29. Kuo, W., & Prasad, V. R., (2000). An annotated overview of system-reliability optimization. *IEEE Transactions on Reliability, 49*(2), 176–187.
30. Lacamera, D., (2018). *Embedded Systems Architecture* (p. 8). Packt. ISBN 978-1-78883-250-2.
31. Lee, C. M., (2000). *The Silicon Valley Edge: A Habitat for Innovation and Entrepreneurship* (Vol. 25, No. 3, pp. 2–8). Stanford University Press.
32. Lee, J. M., & Lee, J. H., (2004). Approximate dynamic programming strategies and their applicability for process control: A review and future directions. *International Journal of Control, Automation, and Systems, 2*(3), 263–278.
33. Linz & Peter (1990). *An Introduction to Formal Languages and Automata* (p. 234). D. C. Heath and Company. ISBN 978-0-669-17342-0.
34. Liskov, B. H., (1972). A design methodology for reliable software systems. In: *Proceedings of the 1972, Fall Joint Computer Conference, Part I* (pp. 191–199).
35. Luus, R., (1975). Optimization of system reliability by a new nonlinear integer programming procedure. *IEEE Transactions on Reliability, 24*(1), 14–16.
36. Martin-Löf, P., (1982). Constructive mathematics and computer programming. In: *Studies in Logic and the Foundations of Mathematics* (Vol. 104, pp. 153–175). Elsevier.
37. McCartney, S., (1999). *ENIAC: The Triumphs and Tragedies of the World's First Computer* (p. 16). Walker and Company. ISBN 978-0-8027-1348-3.
38. Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., Budhia, R. K., & Lingley-Papadopoulos, C. A., (1996). Totem: A fault-tolerant multicast group communication system. *Communications of the ACM, 39*(4), 54–63.

39. Nimmer, R. T., & Krauthaus, P., (1987). Classification of computer software for legal protection: International perspectives. In: *Int'l L.* (Vol. 21, p. 733).

40. Noldus, L. P. J. J., (1991). The observer: A software system for collection and analysis of observational data. *Behavior Research Methods, Instruments, & Computers, 23*(3), 415–429.

41. Park, G. L., Shirazi, B., & Marquis, J., (1997). DFRN: A new approach for duplication-based scheduling for distributed memory multiprocessor systems. In: *Parallel Processing Symposium, International* (pp. 150–157). IEEE Computer Society.

42. Piteira, M., & Costa, C., (2013). Learning computer programming: Study of difficulties in learning programming. In: *Proceedings of the 2013 International Conference on Information Systems and Design of Communication* (pp. 75–80).

43. Riddle, W. E., (1979). An approach to software system behavior description. *Computer Languages, 4*(1), 29–47.

44. Robins, A., Rountree, J., & Rountree, N., (2003). Learning and teaching programming: A review and discussion. *Computer Science Education, 13*(2), 137–172.

45. Rosen, K. H., (1991). *Discrete Mathematics and its Applications* (p. 654). McGraw-Hill, Inc. ISBN 978-0-07-053744-6.

46. Rullan, A., (1997). Programmable logic controllers versus personal computers for process control. *Computers & Industrial Engineering, 33*(1, 2), 421–424.

47. Russell, D. S., Hart, T. P., & Levin, M. (2021). *Lisp (Programming Language).* (Vol. 1, pp. 1–13).

48. Schach, S. R., (1990). *Software Engineering* (p. 216). Aksen Associates Incorporated Publishers. ISBN 0-256-08515-3.

49. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., & Hanrahan, P., (2008). Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics (TOG), 27*(3), 1–15.

50. Shen, K., & Yu, W., (2018). Fractional programming for communication systems—Part I: Power control and beamforming. *IEEE Transactions on Signal Processing, 66*(10), 2616–2630.

51. Silberschatz, A., (1994). *Operating System Concepts* (4th edn., pp. 6, 98). Addison-Wesley. ISBN 978-0-201-50480-4.

52. Sommer, S., Camek, A., Becker, K., Buckl, C., Zirkler, A., Fiege, L., & Knoll, A., (2013). Race: A centralized platform computer-based architecture for automotive applications. In: *2013 IEEE International Electric Vehicle Conference (IEVC)* (pp. 1–6). IEEE.
53. Stair, R. M., (2003). *Principles of Information Systems* (6th edn., p. 159). Thomson. ISBN 0–619-06489-7.
54. Stroustrup, B., (2013). *The C++ Programming Language* (4th edn., p. 40). Addison-Wesley. ISBN 978-0-321-56384-2.s.
55. Tan, S. D., & Shi, C. J., (2003). Efficient very large-scale integration power/ground network sizing based on equivalent circuit modeling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 22*(3), 277–284.
56. Tanenbaum, A. S., (1990). *Structured Computer Organization* (3rd edn., p. 32). Prentice-Hall. ISBN 978-0-13-854662-5.
57. Thota, C., Sundarasekar, R., Manogaran, G., Varatharajan, R., & Priyan, M. K., (2018). Centralized fog computing security platform for IoT and cloud in healthcare system. In: *Fog Computing: Breakthroughs in Research and Practice* (pp. 365–378). IGI global.
58. Tolpygo, S. K., Bolkhovsky, V., Weir, T. J., Wynn, A., Oates, D. E., Johnson, L. M., & Gouker, M. A., (2016). Advanced fabrication processes for superconducting very large-scale integrated circuits. *IEEE Transactions on Applied Superconductivity, 26*(3), 1–10.
59. Turski, W. M., & Wasserman, A. I., (1978). Computer programming methodology. *ACM SIGSOFT Software Engineering Notes, 3*(2), 20–21.
60. Weik, M. H., (1961). The ENIAC story. *Ordnance* (3rd edn., Vol. 45, No. 244, pp. 571–575).
61. Weiss, M. A., (1994). *Data Structures and Algorithm Analysis in C++* (p. 29). Benjamin/Cummings Publishing Company, Inc. ISBN 0-8053-5443-3.
62. Wilson, L. B., (2001). *Comparative Programming Languages* (3rd edn., pp. 7, 29). Addison-Wesley. ISBN 0-201-71012-9.
63. Zhang, M., Ciesielski, V. B., & Andreae, P., (2003). A domain-independent Window approach to multiclass object detection using genetic programming. *EURASIP Journal on Advances in Signal Processing, 2003*(8), 1–19.

64. Zinnat, S. B., (2021). *Classification of Computer Programming Contest Programs Based on Gender, Region, and Software Metrics* (Vol. 1, p. 1–9). Doctoral dissertation, Lethbridge, Alta. University of Lethbridge, Dept. of Mathematics and Computer Science.

CHAPTER 3

FUNDAMENTALS OF PROGRAMMING LANGUAGES

CONTENTS

3.1. INTRODUCTION

A competent programmer may develop good software in every language, just as a good pilot may fly every plane. A passenger plane has been built for luxury, security, and economic feasibility; a military plane is built for performance and mission capabilities, and an ultralight plane is built for cheap cost and ease of operation. Once it is asserted that the well-designed system may be implemented equally effective in every language, the function of language in programming is lowered in favor of software tools and methodologies; not only reduced but completely rejected (Sammet, 1972). However, programming languages are more than simply a tool; they provide the raw resources for software, which is what we spend the majority of our time looking at on our computers. The programming language is among the most essential, if not the most essential, aspects that manipulate the overall quality of a software system. However, several programmers are illiterate. He is enamored with his "native" programming language although is unable to examine and compare language structures, as well as comprehend the benefits and drawbacks of current languages and concepts. "Language L1 is more effective (or efficient) as compared to language L2," for example, is a statement that frequently demonstrates conceptual ambiguity (Rosen, 1971).

Because of this lack of understanding, there are 2 important difficulties in software that must be addressed. For starters, there is an extreme conservatism when it comes to the selection of programming languages. However, despite the rapid advancements in computer technology and the sophistication of current software systems, the vast majority of programming has still been performed in languages that had been invented about 1970, if not before that. Comprehensive programming language study is never put to the test in the real world, and software developers are forced to depend on instruments and approaches to balance for outmoded programming language technology. It's like if airlines will deny experimenting with jet planes on the basis that a traditional propeller aircraft is completely able to transport passengers from point A to point B just as efficiently (Sammet, 1991).

In addition, language structures are employed arbitrarily, with no or little concern for the security or effectiveness of the system. This results in faulty software that may not be sustained, and also inefficiencies that are rectified through assembly language coding instead of by improvement of the programming paradigms and algorithms themselves (King, 1992).

It is solely for the aim of bridging the gap in the level of abstraction among the actual world and hardware that programming languages have been created. An inevitability exists between greater levels of abstraction which are simpler to comprehend and secure for usage on the one hand and relatively low levels of abstraction which are more adaptable and may frequently be implemented more effectively on the other hand. To create or adopt a programming language, one must first determine the proper degree of abstraction to use. It is not unusual that various programmers like various levels of abstraction, or that a certain language can be suitable for one project but not for the next. A programmer must be well-versed in the efficiency and safety implications of every construct in a particular programming language in which they are working (Mészárosová, 2015).

3.2. PURPOSE OF PROGRAMMING LANGUAGES

When learning the latest programming language, the question that comes to mind most often is : "What may such language "do?"

We have been inadvertently evaluating the modern language to certain other languages. The solution is straightforward: all languages are capable of performing similar calculations! The reasoning for this response is outlined in the following section. There should be other principal causes of many programming languages whether they may all perform similar calculations (Heim et al., 2020).

Let's begin with a few definitions:

"*A program is a set of secret code that describes how to do a calculation. A programming language is a collection of rules that describe which symbol sequencing make up a program as well as what calculation it does*" —(Kip-er et al., 1997).

It's worth noting that the definition makes no mention of the term computer! Languages and p Program are mathematical objects that are fully formal. Furthermore, many consumers are more curious about programs as compared in other mathematical objects like groups, whereas a program is a series of symbols that may be used to control a computer's execution. While studying the theory of programming is highly recommended, this course would primarily focus on the study of programs since they are performed on a computer (Davison et al., 2009).

These are fairly broad concepts that must be construed as widely as possible. For instance, most advanced word processors contain a feature that allows you to "catch" a series of keystrokes and save them as a macro, allowing you to run the entire sequence with a single keystroke. Since the sequence of keystrokes indicates a computation, and the software handbook would clearly describe the macro language: how to terminate, launch, and identify a macro definition, this is unquestionably a program (Atkinson and Buneman, 1987).

To address the issue posed at the beginning of this section, we must first return to primitive digital computers, that are similar to the rudimentary calculators utilized through your grocer nowadays in that the calculation performed by these computers is "wired-in" and it may not be modified.

The revelation (attributed to John von Neumann) that the definition of the calculation, the program, may be accumulated in the computer almost as readily as the data utilized in the computation had been the most important early achievement in computers. As a result, the stored-program computer becomes a basic-purpose computing machine, and we may alter the program simply via inserting the punched card, changing the plugboard of wiring, linking to a phone line, or adding a diskette (Figure 3.1) (Davison et al., 2009).

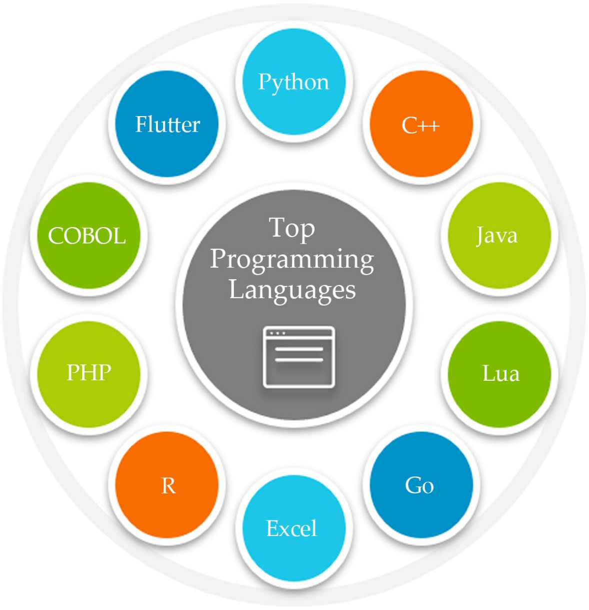


Figure 3.1. Main programming languages.

Source: https://www.bmc.com/blogs/programming-languages/.

Because computers are binary processors that only recognize zeros and ones, maintaining programs in them is mathematically simple but difficult in practice, because every command must be recorded as binary digits (bits) that may be shown electronically or mechanically. The symbolic assembler had been the one of the first software tools developed to solve this issue. An assembler analyzes an assembly language program that represents every instruction like a symbol and converts it to a binary form appropriate for computer execution. For instance, consider the following instruction (Ahmed et al., 2014):

```
load    R3,54
```

Its meaning is far more comprehensible than the corresponding string of bits: "load register 3 having the information in memory location 54." Believe this or not, the phrase "automated programming" initially applied to assemblers, which chose the correct bit sequence for every symbol automatically. Pascal and C are more advanced as compared to assemblers since they "automatically" select registers and addresses, as well as "automatically" select instruction orders to construct arithmetic expressions and loops (Antolík and Davison, 2013).

We're now able to respond to the question posed at the start of this section.

A programming language is a method for abstraction. It allows a programmer to abstractly express a computation and then have a program (typically referred to as an interpreter, compiler, or assembler) execute the specification in the exact format required for computer performance.

It may also see why there are various programming languages: there are two distinct types of issues can necessitate various abstraction levels, and various programmers can have various opinions about how abstraction must be accomplished. A "C" programmer has been quite pleased to operate at an abstraction level that necessitates the definition of calculations utilizing indices and arrays, but a report author likes to "program" utilizing a language made up of word-processor functions (Vella et al., 2014).

The degrees of abstraction in computer hardware may be seen clearly. Separate components like resistors and transistors were linked directly at first. Then simple plug-in modules and smaller-scale ICs were used. Currently, whole computers may be manufactured from only a few chips, every one of which has many components. No computer specialist would try to construct a "perfect" circuit from single parts if a group of chips that could be changed to fulfill the same function existed (Flatt et al., 1999).

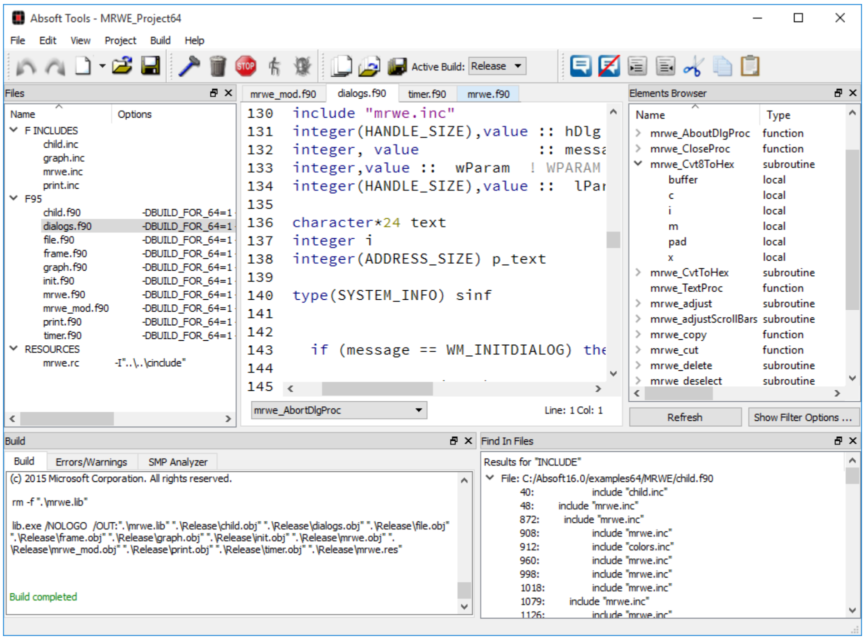
The idea of abstraction generates a universal truth—more information is lost as the abstraction level rises. While writing a program in C, you lose the capability to describe register allocation that you had in assembly language; when writing in Prolog, you lose the capability to express random connected structures utilizing pointers that you had in C. There's a natural conflict between desiring the freedom of expressing the calculation in detail and seeking for a succinct, unambiguous, and trustworthy description of a calculation in a higher-level abstraction. A lower-level description will always be more precise and optimum than an abstraction (Wasserman and Prenner, 1979).

Starting with "common" programming languages such as FORTRAN, Pascal, C, and the Pascal-like features of Ada, we would cover languages at 3° of abstraction in this chapter. Finally, in Part IV, we'll look at languages like C++ and Ada, which allow programmers to create high-level abstractions from simple statements. Ultimately, we'll talk about logical and functional programming languages, which operate at even high abstraction levels (Fourment and Gillings, 2008).

3.3. IMPERATIVE LANGUAGES

3.3.1. FORTRAN

FORTRAN was the 1st language of programming that advanced substantially beyond assembly code. It had been created via an IBM team has led by John Backus in the 1950s to give an abstract manner of defining scientific calculations. FORTRAN faced stiff resistance for the same reasons that all succeeding ideas for high-level abstractions did: many programmers thought that a compiler might not create optimum code when compared to hand-coded assembly language (Figure 3.2) (Ottenstein et al., 1990).



```
Name              Options
F INCLUDES
    child.inc
    graph.inc
    mrwe.inc
    print.inc
F95
    child.f90     -DBUILD_FOR_64=1
    dialogs.f90   -DBUILD_FOR_64=1
    file.f90      -DBUILD_FOR_64=1
    frame.f90     -DBUILD_FOR_64=1
    graph.f90     -DBUILD_FOR_64=1
    init.f90      -DBUILD_FOR_64=1
    mrwe.f90      -DBUILD_FOR_64=1
    mrwe_mod.f90  -DBUILD_FOR_64=1
    print.f90     -DBUILD_FOR_64=1
    timer.f90     -DBUILD_FOR_64=1
RESOURCES
    mrwe.rc       -I"..\..\include"
```

```
include "mrwe.inc"
integer(HANDLE_SIZE),value :: hDlg
integer, value             :: messa
integer,value ::  wParam  ! WPARAM
integer(HANDLE_SIZE),value ::  lPar

character*24 text
integer i
integer(ADDRESS_SIZE) p_text

type(SYSTEM_INFO) sinf


  if (message == WM_INITDIALOG) the
```

```
Name                     Type
mrwe_AboutDlgProc        function
mrwe_CloseProc           function
mrwe_Cvt8ToHex           subroutine
    buffer               local
    c                    local
    i                    local
    m                    local
    pad                  local
    x                    local
mrwe_CvtToHex            subroutine
mrwe_TextProc            function
mrwe_adjust              subroutine
mrwe_adjustScrollBars    subroutine
mrwe_copy                function
mrwe_cut                 function
mrwe_delete              subroutine
mrwe_deselect            subroutine
```

```
(c) 2015 Microsoft Corporation. All rights reserved.

 rm -f ".\mrwe.lib"

 lib.exe /NOLOGO  /OUT:".\mrwe.lib" ".\Release\child.obj" ".\Release\dialogs.obj" ".\Release\file.obj"
".\Release\frame.obj" ".\Release\graph.obj" ".\Release\init.obj" ".\Release\mrwe.obj" ".
\Release\mrwe_mod.obj" ".\Release\print.obj" ".\Release\timer.obj" ".\Release\mrwe.res"


Build completed
```

```
Results for "INCLUDE"
File: C:/Absoft16.0/examples64/MRWE/child.f90
    40:            include "child.inc"
    48:       include "mrwe.inc"
    872:       include "mrwe.inc"
    908:           include "mrwe.inc"
    912:           include "colors.inc"
    960:           include "mrwe.inc"
    998:           include "mrwe.inc"
    1018:          include "mrwe.inc"
    1079:      include "mrwe.inc"
    1126:          include "mrwe.inc"
```

Figure 3.2. Windows Fortran compiler suite.

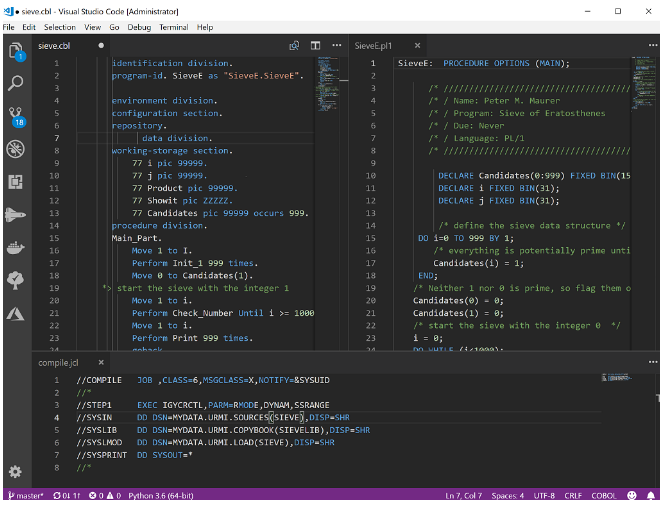
Source: https://www.absoft.com/products/windows-fortran-compiler-suite/.

FORTRAN, like other early programming languages, had severe flaws, both in terms of language construction and support for module organizing notions and current data. In retrospect, Backus remarked, "Which as it had been known already, we just built up the language whenever we went alongside." We didn't see design of language as a challenging task, but rather as a straightforward prolog to the main challenge: creating an effective compiler (Cann, 1992).

Nonetheless, the benefits of abstraction rapidly won over so many programmers: fast and reliable design, as well as reduced machine reliance due to the abstraction of register and machine instructions. FORTRAN had become the standard language in research and engineering as most early computers had been focused on scientific issues, and it is only now being supplanted by newer languages. FORTRAN has been extensively modernized (in 1966, 1977, 1990) to meet the needs of current software development (Burgess and Saidi, 1996).

3.3.2. COBOL and PL/I

The COBOL programming language had been created for commercial data processing in the 1950s. A group comprised of members from the United States Defense Department, commercial entities, computer manufacturers and like insurance firms drafted the phrase. COBOL had been aimed to be a temporary solution until an improved design might be developed; rather than, the language just like described quickly had become the widely used language in its sector (much like FORTRAN in science), and for the same reason: it gives a better resource of expressing calculations which are common in its area. The requirement to perform relatively basic computations on large numbers of complicated data records characterizes business data processing, and COBOL's data structuring abilities greatly outstrip that of algorithmic languages such as C or FORTRAN (Figure 3.3) (Lorenzen, 1981).



sieve.cbl - Visual Studio Code [Administrator]
File Edit Selection View Go Debug Terminal Help
sieve.cbl
SieveE.pl1
compile.jcl

```
identification division.
program-id. SieveE as "SieveE.SieveE".

environment division.
configuration section.
repository.
        data division.
working-storage section.
    77 i pic 99999.
    77 j pic 99999.
    77 Product pic 99999.
    77 Showit pic ZZZZZ.
    77 Candidates pic 99999 occurs 999.
procedure division.
Main_Part.
    Move 1 to I.
    Perform Init_1 999 times.
    Move 0 to Candidates(1).
*> start the sieve with the integer 1
    Move 1 to i.
    Perform Check_Number Until i >= 1000
    Move 1 to i.
    Perform Print 999 times.
```

```
SieveE:  PROCEDURE OPTIONS (MAIN);

      /* ///////////////////////////////////
      /* / Name: Peter M. Maurer
      /* / Program: Sieve of Eratosthenes
      /* / Due: Never
      /* / Language: PL/1
      /* ///////////////////////////////////

        DECLARE Candidates(0:999) FIXED BIN(15
        DECLARE i FIXED BIN(31);
        DECLARE j FIXED BIN(31);

        /* define the sieve data structure */
    DO i=0 TO 999 BY 1;
       /* everything is potentially prime unti
       Candidates(i) = 1;
    END;
   /* Neither 1 nor 0 is prime, so flag them o
   Candidates(0) = 0;
   Candidates(1) = 0;
   /* start the sieve with the integer 0  */
   i = 0;
```

```
//COMPILE   JOB ,CLASS=6,MSGCLASS=X,NOTIFY=&SYSUID
//*
//STEP1     EXEC IGYCRCTL,PARM=RMODE,DYNAM,SSRANGE
//SYSIN     DD DSN=MYDATA.URMI.SOURCES(SIEVE),DISP=SHR
//SYSLIB    DD DSN=MYDATA.URMI.COPYBOOK(SIEVELIB),DISP=SHR
//SYSLMOD   DD DSN=MYDATA.URMI.LOAD(SIEVE),DISP=SHR
//SYSPRINT  DD SYSOUT=*
//*
```

master* 0↓ 1↑ 0 0 Python 3.6 (64-bit)
Ln 7, Col 7 Spaces: 4 UTF-8 CRLF COBOL

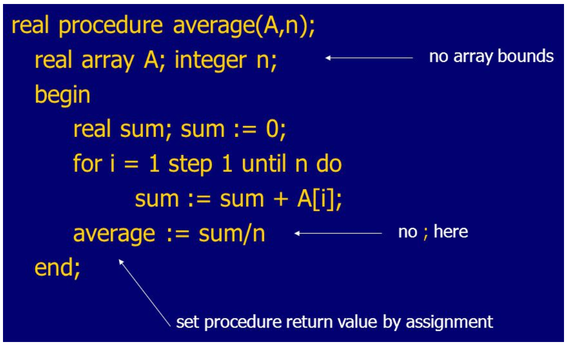
Figure 3.3. Code colorization for PL/I and COBOL.

Source: https://marketplace.visualstudio.com/items?itemName=bitlang.cobol.

Afterwards, IBM developed PL/I as a worldwide language that included the characteristics of COBOL, Algol, and FORTRAN. On several IBM systems, PL/I have supplanted COBOL and FORTRAN, however, this huge language had never been generally supported outside of IBM, particularly on the microcomputers and minicomputers that are becoming incredibly common in the processing of data firms (Heller and Logemann, 1966).

3.3.3. Algol and Its Descendants

Algol has had the most effect on the language design of all the earlier programming languages. It was created through an international team for generic and scientific purposes, but due to FORTRAN's strong backing from major computer manufacturers, it never gained mainstream acceptance. The initial version of Algol had been released in 1958, and the updated version, Algol 60, had been widely utilized in computer science research and was installed on a large number of computers, particularly in Europe. Algol 68, a 3rd version of the language, was significant among language theorists but was never extensively used (Figure 3.4) (Wijngaarcien et al., 1977).



```
real procedure average(A,n);
  real array A; integer n;
  begin
     real sum; sum := 0;
     for i = 1 step 1 until n do
          sum := sum + A[i];
     average := sum/n
  end;
```

no array bounds
no ; here
set procedure return value by assignment

Figure 3.4. Sample syntax of Algol language.

Source: https://slideplayer.com/slide/2368014/.

Jovial, which is utilized through the Air Force of United States for real-time systems, and Simula, one of the earliest simulation languages, are 2 prominent languages that had been evolved from Algol. Pascal, invented through Niklaus Wirth in the late 1960s, is possibly the most renowned descendant of Algol. Pascal had been born out of a desire to build a language that might be utilized to teach concepts such as type checking and type declarations (McCusker, 2003).

Pascal has one major benefit and one major shortcoming as a practical language. Because the first Pascal compiler had been written in Pascal, it was simple to transfer to any machine. The language spread swiftly, particularly among the microcomputers and minicomputers that were being developed at the time. Regrettably, the Pascal programming language is just excessively limited. The standard language has no way of breaking a program into modules on distinct files, therefore it can't be utilized to write programs with more than a few thousand lines. Although practical Pascal compilers enable module deconstruction, there is no standard mechanism; therefore, huge applications are not portable (Valverde and Solé, 2015).

Wirth realized the need for modules in every practical language and created the Modula language as a result. Modula has a famous option to non-standard Pascal dialects (currently in version 3 with support for object-oriented programming (OOP)) (Ginsburg and Rose, 1963).

Dennis Ritchie of Bell Laboratories created the C programming language in the starts of 1970s as an implementation language for the operating system (OS) of UNIX. As higher-level languages had been deemed wasteful, OSs had been usually developed in assembly code. By providing data structures and structured control statements (records and arrays), the C abstracts away the complexities of assembly language programming whilst retaining all of the flexibility of lower-level programming in assembly language (bit-level operations and pointers) (Reddy, 2002).

UNIX soon became the choice system in research and academic institutions because it had been freely available to universities and had been designed in a portable language instead of raw assembly code. When modern computers and programs came out of these universities and into the commercial sphere, they brought UNIX and C with them.

Because harmful constructs aren't examined via the compiler, C is supposed to be as versatile as assembly language. The difficulty is that this flexibility makes it very simple to develop programs with cryptic problems. When it is used correctly on tiny programs, the C is a precise language, but when utilized on huge software systems produced via teams of varied abilities, it may cause major problems. Several of the hazards of constructions in C would be discussed, as well as how to avoid key mistakes (Yang et al., 2006).

The American National Standards Institute (ANSI) standardized the C programming language in 1989, and the International Standards Organization (ISO) approved virtually the same standard a year later. The C in this book refers to ANSI C rather than older versions of the language.

3.3.4. C++

Bjarne Stroustrup, also of Bell Laboratories, created the C++ language in the 1980s, expanding C to incorporate OOP features comparable to that of the Simula language. Furthermore, C++ corrects numerous errors in C and must be utilized instead of C in tiny applications where object-oriented capabilities aren't required. When updating a C-based system, C++ is the natural language to utilize (Figure 3.5) (Ishikawa et al., 1996).

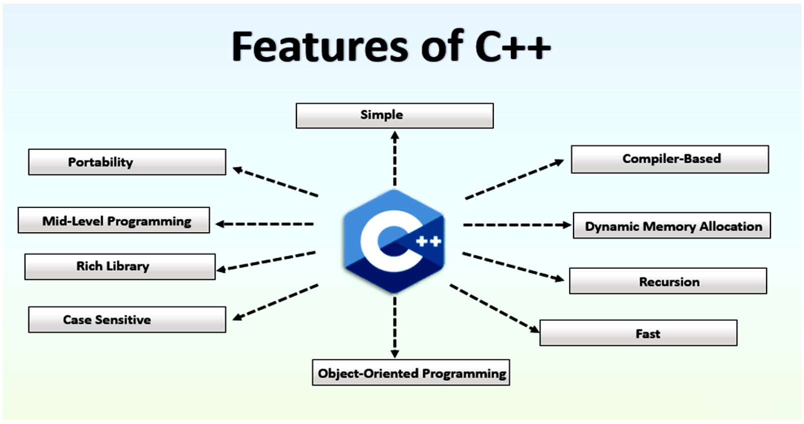


Figure 3.5. Salient features of C++.

Source: https://www.educba.com/features-of-c-plus-plus/.

Please keep in mind that C++ is a dynamic language, thus your reference compiler or manual cannot be completely up to date.

3.3.5. Ada

The US Department of Defense decided to standardize on a single programming language in 1977, mostly to keep money on training and on the expense of sustaining program creation environments for every system of military, according to the official history. Following an evaluation of current languages, they decided to request the development of a novel language that would be dependent upon a competent existing language, like the Pascal programming language. Ultimately, one of the proposals for a language had been selected and named Ada, and a standard had been established in 1983. Ada is exceptional in various ways (Sward et al., 2003):

- A single team created and developed the majority of programming languages (Pascal, C, FORTRAN, etc.), and they had been only standardized after being widely used. All of the unintentional mistakes made by the original teams had been included in the standard for the sake of compatibility. Ada was exposed to extensive study and criticism before being standardized.

- Most programming languages were first built on one computer and had been substantially impacted through the oddities of that computer before being standardized. Ada had been created to facilitate the creation of portable applications.
- In addition, Ada broadens the scope of programming languages through allowing the handling of error and concurrent programming, both of which have previously been reserved for (non-standard) OS functions (Figure 3.6).

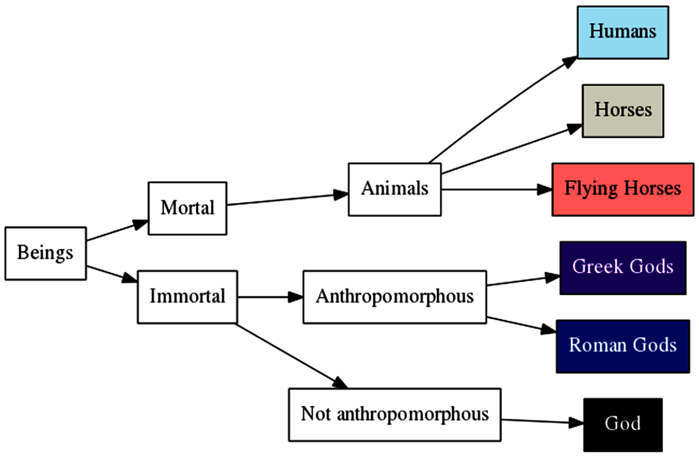


Figure 3.6. The object-oriented paradigm of Ada language.

Source: https://peakd.com/ada-lang/@xinta/learning-ada-5-object-oriented-paradigm.

Although its technological superiority and the benefits of early standardization, Ada has been unable to gain general acceptance exterior of military and larger-scale applications (like commercial aviation and transportation by rail). Ada has a repute for being a tough language. It has been because the language covers several areas of programming those other languages (such as Pascal and C) leave to the OS, therefore there is just more to learn. In addition, better, and affordable educational development settings weren't readily accessible. Ada is becoming more widely utilized in the academic curriculum, although as a "primary" language, thanks to the availability of free compilers and solid introductory textbooks (Hutcheon and Wellings, 1988).

3.3.6. Ada 95

A new standard for the Ada language is issued exactly 12 years after the initial standard for the Ada language was finalized in 1983. The latest version, dubbed Ada 95, fixes a few flaws in the previous version. The most significant addition is support for real OOP, including inheritance, which had been left out of Ada 83 due to perceived inefficiency. Annexes to the Ada 95 standard define standard (although optional) additions for information systems, real-time systems, secure systems, numeric's, and distributed systems (Bailes, 1992).

If the subject is exclusive to single version: "Ada 95" or "Ada 83," the name "Ada" would be used in this text. Because the actual year of standardization was unknown during development, Ada 95 had been referred to as Ada 9X in the literature.

3.4. DATA-ORIENTED LANGUAGES

Many notable languages had been conceived and implemented in the initial days of programming, all of which shared one feature: every language had a chosen data structure and a comprehensive operations set for that structure. Such languages allowed programmers to develop sophisticated programs that would have been impossible to write in languages like FORTRAN, which only handled computer text. We'll look at a few of such languages in more detail in the subsections that follow (Denning, 1978).

3.4.1. Lisp

The linked list is the most fundamental data structure in Lisp. Significant work on artificial intelligence (AI) had been done in Lisp, which had been created for study in computation theory. Because the language was so vital, machines were created and built specifically to run Lisp applications (Figure 3.7) (Murphree and Fenves, 1970).

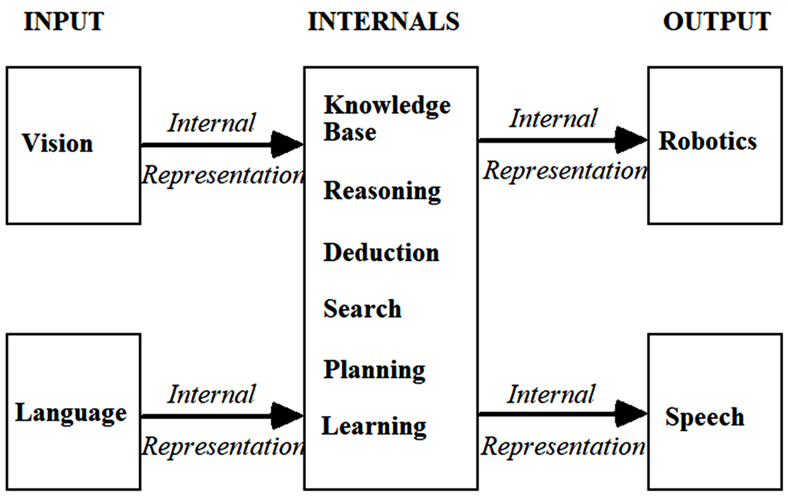


Figure 3.7. Artificial intelligence utilizing lisp programming.

Source: https://www.electroniclinic.com/artificial-intelligence-using-lisp-programming-examples/.

The growth of numerous dialects when the language had been implemented on various devices was one issue with the language. Subsequently, the Common Lisp programming language had been created to allow applications to be transferred from one machine to another. CLOS, a prominent dialect of Lisp that allows OOP, is now a famous dialect.

Cdr(L) and car (L), which remove the tail and head of a list L, correspondingly, and cons(E, L), which builds a fresh list from a component E and an old list L, are the 3 basic Lisp operations. Functions to processing lists comprising non-numeric data may be constructed utilizing these techniques; these functions will be exceedingly complex to implement in FORTRAN (Rajaraman, 2014).

Lisp is a long-lived programming language that has been in use for about a quarter-century. Just FORTRAN has a longer history amongst active programming languages. Both languages have met the programming requirements.

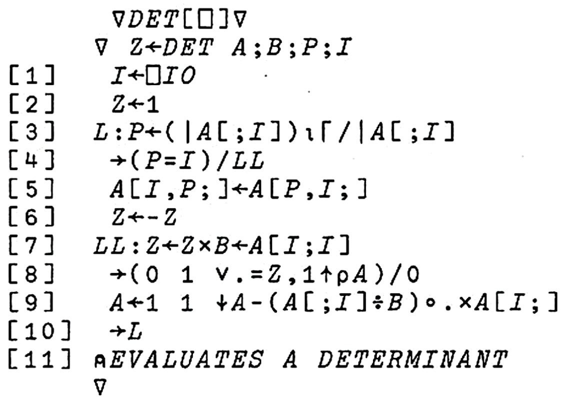
FORTRAN for scientific and technical calculation and Lisp for AI are two prominent areas of application. These 2 fields are still vital, and their programmers are so dedicated to such 2 languages that FORTRAN and Lisp may stay in usage for another quarter-century.

AI research, as one might assume given its aims, creates a slew of serious programming issues. This rash of difficulties has spawned new languages in different programming cultures. Likewise, controlling, and isolating traffic inside work modules by the development of language is a valuable organizational approach in any extremely big programming effort. As one reaches the limits of the system wherein, we humans interact more frequently, such languages start to become less rudimentary (Adeli and Paek, 1986).

As a result, these systems have several copies of complicated language-processing functions. Because Lisp's semantics and syntax are so basic, parsing might be considered a trivial process. As a result, parsing technology plays essentially no part in Lisp programs, and the development of language processors is seldom a hindrance to the rate at which big Lisp systems expand and evolve. Ultimately, it is the freedom and burden that all Lisp programmers bear because of the simplicity of syntax and semantics. There is no way to write a Lisp program larger than several lines without using discretionary functions (Swift and Mize, 1995).

3.4.2. APL

The APL programming language arose from a mathematical notation for describing computations. Matrices and vectors are the most fundamental data structures. Operations are performed directly on them without the use of loops. As a result, when compared to equivalent programs written in other languages, the programs are extremely brief. One issue with APL is that it retains a huge number of mathematical signs from basic formalism. This necessitates the usage of a particular terminal, making it impossible to test with APL with no investing in expensive hardware; newer graphical user interfaces (GUIs) that employ fonts of software have eliminated such difficulty, hastening APL's adoption (Figure 3.8) (McIntyre, 1991).



```
           ∇DET[⎕]∇
          ∇ Z←DET A;B;P;I
    [1]    I←⎕IO
    [2]    Z←1
    [3]   L:P←(|A[;I])⍳⌈/|A[;I]
    [4]    →(P=I)/LL
    [5]    A[I,P;]←A[P,I;]
    [6]    Z←-Z
    [7]   LL:Z←Z×B←A[I;I]
    [8]    →(0 1 ∨.=Z,1↑⍴A)/0
    [9]    A←1 1 ↓A-(A[;I]÷B)∘.×A[I;]
    [10]   →L
    [11]  ⍝EVALUATES A DETERMINANT
          ∇
```

Figure 3.8. Sample syntax of APL language.

Source: https://computerhistory.org/blog/the-apl-programming-language-source-code/.

3.4.3. Snobol, Icon

Numbers were almost solely dealt with in earlier languages. Snobol (and its descendant Icon) are appropriate for work in disciplines like natural language processing since its core data structure is the string. Snobol's main operation is to match a pattern to a string, with the string being deconstructed into substrings as a result of the match. Expression assessment is the most fundamental process in Icon, although expressions can encompass complicated string manipulations (Jeffery et al., 2016).

The find(s1, s2) is a useful predefined function in Icon that looks for instances of the string s1 in the string s2. Find produces a list of all spots in s2 where s1 appears, unlike an equivalent function in C:

```
line:= 0# Initialize line counter
while s:= read() {        # Read until end of file every col:= find("the", s) do
          # Generate column positions
write(line, " ", col)       # Write (line,col) of "the"
line:= line + 1}
```

The column and line numbers of all locations of the string "the" in a file would be written by this application. If the search fails to identify an occurrence, the expression's computation is ended. The keyword each compels the function to be evaluated again as long as this is effective.

Icon expressions may be described on csets, that are sets of characters, as well as strings, that are sequences of characters. Thus:

vowels:= ‘aeiou’

It assigns a value to the variable vowel, which is the set of letters shown. This is utilized in methods such as up to (vowels,s), which returns the longest beginning order of vowels in s, and several (vowels,s), that returns the order of positions of vowels in s.

Bal is very complicated function that works similar up to which it creates orders of locations that are balanced in terms of bracketing characters:

bal(‘+–*/,’ ‘([,’ ‘)],’ s)

This expression might be utilized to construct balanced arithmetic sub-strings in a compiler. Given the string “x+(y[u/v]–1)*z,” The indices relating to the sub-strings would be generated by the above equation:

x x + (y[u/v] – 1)

The 1st sub-string has been balanced since this end with “+” and has no bracketing characters; the 2nd sub-string is balanced since it ends with “*” and has square brackets properly contained within parentheses.

Backtracking may be utilized to resume the search from previous generators if an expression fails. Except for those that begin in column 1, the following software prints the appearances of vowels:

```
line:= 0      # Initialize line counter
while s:= read() {   # Read until end of file every col:= (up to(vowels, line) >1) do
         # Generate column positions
write(line, “ ”, col) # Write (line,col) of vowel
line:= line + 1}
```

The function ‘find’ creates an index, which is subsequently checked via “¿.” If the experiment is not successful (do not state “if the outcome is false”), the program goes back to the generator function and requests a newer index.

The icon is a useful language for applications that need to manipulate strings in a complicated way. The majority of the explicit calculation using indices is abstracted away, resulting in highly compact programs as compared to standard languages meant for numerical or programming of systems. The icon is also intriguing as of the built-in generation and backtracking mechanisms, that provide an additional degree of control abstraction (Kennedy and Schwartz, 1975).

3.4.4. SETL

The set is the most fundamental data structure in SETL. SETL may be utilized to generate generalized programs that are highly abstract and hence very brief because sets have been the most generic mathematical structure by which all mathematical structures have been created. In the sense that mathematical descriptions may be directly executed, the programs are similar to logic programming. Set theory notation is utilized: $\{x \mid p(x)\}$, which denotes the set of all x for whom the logical expression $p(x)$ is true. A mathematical specification of the prime numbers set, for instance, maybe phrased as follows:

$\{n \mid \neg\exists m[(2 \leq m \leq n - 1) \wedge (n \bmod m = 0)]\}$

This formula is written as follows: the set of integers such that no number m among 2 and $n - 1$ divides n without leaving a remainder.

We simply interpret the description into a one-line SETL program to print all primes in the range 2 to 100:

```
print({n in {2.100} — not exists m in {2.n–1} — (n mod m) = 0});
```

Essentially, all such languages approach creation from a mathematical standpoint, asking how may an understanding theory be executed, instead of from an engineering standpoint, asking how may instructions be given to the memory and CPU. These sophisticated programming languages are extremely beneficial for tough programming jobs when it has been critical to concentrate on the issue rather than on lower-level aspects such as syntax and semantics (Dubinsky, 1995).

Data-oriented languages are not very much famous as compared to they once were, owing to competition from new language approaches like functional and logical programming, as well as the ability to integrate these data-oriented processes into regular languages such As C++ and Ada utilizing object-oriented approaches. Nonetheless, the languages are both technically fascinating and extremely useful for the programming tasks for which they had been created. Students must try to learn at least one of such languages since they expand their understanding of how a programming language might be organized (Grove et al., 1997).

3.5. OBJECT-ORIENTED LANGUAGES

OOP is a way of arranging programs that involves recognizing real-world or other objects and then building modules that comprise all the information and readable statements required to show a certain class of objects. There is a clear separation inside such a module between the class's abstract characteristics that are exposed for usage through other objects and the execution that is concealed so that it may be changed with no impact on the remaining system (Figure 3.9) (Blanchet, 1999).

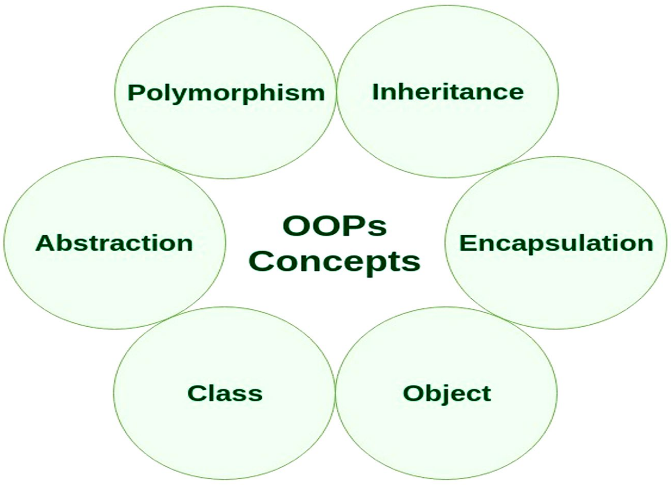


Figure 3.9. Main concepts in object-oriented programming.

Source: https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/.

Simula, the 1st OOP language, had been developed by K. Nygaard and O.J. Dahl in the 1960s for the simulation of system: every sub-system participating in the simulation had been written as an object. Because every subsystem might have several instances, a class may be created to describe every subsystem, and objects of this type may then be allocated (Ferber, 1989).

With the Smalltalk programming language, the Xerox Palo Alto Research Center promoted OOP. The same research gave birth to today's popular windowing systems, and one of Smalltalk's biggest advantages is that it is not just a language, although an entire programming environment. Smalltalk's technological breakthrough was to demonstrate that a language may be created using simply objects and classes as structural structures, negating the necessity to bring these notions into a "normal" language.

Operation dispatching, Allocation, and type checking are dynamic (run-time) rather than static (static) in such pioneering object-oriented languages, which has hampered widespread acceptance of OOP (compile-time). Without getting into specifics, the effect is that programs written in such languages have a memory and time overhead that may be exorbitant in several kinds of systems. Furthermore, static sort checking is increasingly seen as critical for the development of trustworthy software. As a result, Ada 83 only included a portion of the language features needed for OOP (Snyder, 1986).

C++ demonstrated that the full OOP machinery can be implemented in a way that is compatible with type-checking and static allocation, as well as fixed overhead for dispatching; the dynamic needs of OOP are only utilized as required. Ada 95's OOP support was built on concepts comparable to those present in C++.

To get these benefits, although, it is not required to splice OOP support onto current languages. The Eiffel language is comparable to Smalltalk in that classes and objects are the sole way to structure code, and it is alike to Ada 95 and C++ in that it has been statically type-checked and object execution may be dynamic or static depending on the situation. Eiffel is a fantastic choice for a first programming language because of its simplicity in comparison to "hybrids" and complete support for OOP. Java is both a programming language and a framework for creating network software. The syntax is similar to C++, but the semantics are significantly different since, such as Eiffel, this is a "pure" OO language that needs robust type checking (America and Linden, 1990).

We will go through OOP language support in Java, C++, and Ada 95 in great depth. A brief introduction of Eiffel would also demonstrate what a "pure" OOP language looks like.

3.6. NON-IMPERATIVE LANGUAGES

All the programming languages that we have covered having one thing in common: the assignment statement, that instructs the computer to transfer information from one location to another, is their fundamental statement. This is a rather modest degree of abstraction when compared to the number of abstractions required to address the issues we wish to tackle through the use of computing. Modern programming languages prefer to define an issue and then leave it to the computer to find out how to resolve it, instead of detailing in greater depth how to transfer information from one location to another (Kumar and Wyatt, 1995).

Newer software packages are composed of computer languages that are quite abstract. It is possible to define a sequence of database structures and screens using an application generator, as well as the generator will then automatically generate the lower-level instructions required to implement the program. Similarly, simulation programs, desktop publishing software, spreadsheets, and other similar applications provide substantial abstraction programming capabilities. However, one downside of this form of software is that it is typically restricted in terms of the kinds of applications that may be readily programmed. In the perspective that you may customize the package to run the program you require simply by the supply of descriptions as parameters, it seems logical that they are referred to as parameterized programs (Figure 3.10) (Raihany and Rabbianty, 2021).

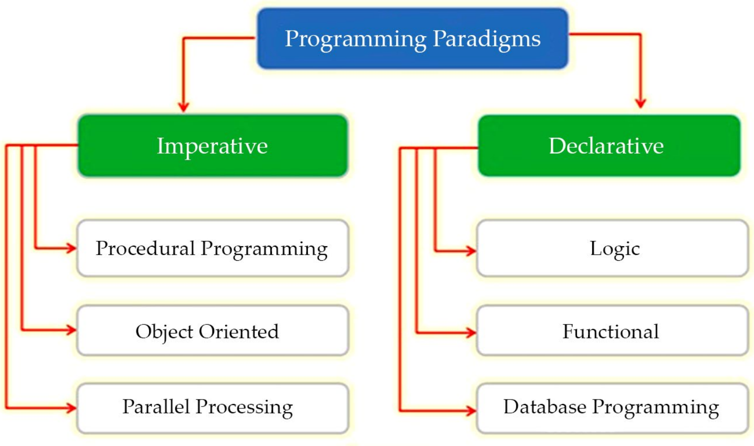


Figure 3.10. Programming language paradigms.

Source: https://www.learncomputerscienceonline.com/computer-programming/.

Another way to express a computation in abstract programming is to use logical implications, functions, equations, or any other formalism. Because mathematical formalisms are employed, these languages are truly basic-purpose programming languages that are not restricted to a single application domain. The compiler does not convert the program into machine code;

alternatively, it tries to resolve a mathematical problem, the answer of which is regarded as the program's outcome. Such programs may be an order of magnitude less than typical programs since loops, pointers, indices, and other details are taken out. The fundamental issue with descriptive programming is that computational operations like I/O to a screen or disc do not fit well with the paradigm, necessitating the use of standard programming tools (Aguado and Pine, 2002).

There are two non-imperative language formalisms that we will explore programming may be divided into two types of programming:

- **Functional Programming:** Programming that is depending upon the mathematical principles of pure functions, such as log and sin, which do not reconfigure their environments, in contrast to so-called functions in an ordinary language such as C, which may have drawbacks;
- **Logic Programming:** Programs are demonstrated as formulas in mathematical logic, and the "compiler" tries to interpret the logical reasoning of such formulas to resolve issues.

Programs written in an abstract, non-imperative language may not aspire to be as effective as hand-coded C programs. However, this is not the case. When a software system should search through enormous volumes of data or resolve issues whose answer may not be explicitly specified, non-imperative languages should be used instead of imperative ones. Pattern matching (genetics, vision), Language processing (style checking, translation), and optimization of the process are all instances of AI (scheduling). It is expected that these languages would become more popular as implementation methods improve and it becomes increasingly hard to construct dependable software systems in traditional programming languages (Jones, 2004).

It is strongly suggested that students learn to program in logical and functional programming languages as their 1st programming languages so that they learn how to work at high degrees of abstraction from the beginning as compared if they had been introduced to programming through C or Pascal.

3.7. STANDARDIZATION

The significance of standardization cannot be overstated. Programs may be translated from one machine to another if a standard for the language exists and compilers follow it. If you are building software that will work on a variety of systems, you should follow a set of guidelines. However, keeping track of dozens or even hundreds of computer-specific elements would make your maintenance duty incredibly difficult (Rao et al., 2021).

For most of the languages covered here, standards are available (or are in the works). Regrettably, the standards had been submitted years after the languages gained popularity and therefore should maintain computer-specific peculiarities from premature executions. The language of Ada is unique in that the standards (1983 and 1995) had been developed and assessed concurrently with the language's design and execution. Moreover, the standard is maintained, allowing compilers to be compared primarily on cost and performance instead of standard conformance. Other languages' compilers can feature a mode that warns you if you use a non-standard construct. If these constructions are required, they must be contained in a small number of well-documented modules (Patel et al., 2022).

3.8. COMPUTABILITY

Logicians researched abstract principles of computing in the 1930s, long before digital computers had been conceived. Both Alan Turing and Alonzo Church created exceedingly basic models of computing (referred to as Turing machines and Lambda calculus, correspondingly), and subsequently established the Church-Turing Thesis (Kari and Thierrin, 1996):

In one of these models, you may do any useful calculation.

Turing machines are relatively basic; there are only two data declarations in C syntax:

```
char tape[…]; int current = 0;
```

Wherein the tape has the ability to go on forever. A program is made up of every number of statements of the following format:

```
L17: if (tape[current] == 'g') {tape[current++] = 'j'; go to L43;}
```

A Turing machine's statement is executed by the following stages:

- Read and inspect the current character on the tape's current cell;
- Replace the character with a different one (optional);
- Increase or decrease the current cell's pointer.

REFERENCES

1. Adeli, H., & Paek, Y. J., (1986). Computer-aided design of structures using LISP. *Computers & Structures, 22*(6), 939–956.
2. Aguado-Orea, J., & Pine, J. M., (2002). There is no evidence for a 'no overt subject' stage in early child Spanish: A note on Grinstead (2000). *Journal of Child Language, 29*(4), 865–874.
3. Ahmed, F. Y., Yusob, B., & Hamed, H. N. A., (2014). Computing with spiking neuron networks a review. *International Journal of Advances in Soft Computing & its Applications, 6*(1), 1–14.
4. Ali, M. S., Babar, M. A., Chen, L., & Stol, K. J., (2010). A systematic review of comparative evidence of aspect-oriented programming. *Information and software Technology, 52*(9), 871–887.
5. Alkhatib, G., (1992). The maintenance problem of application software: An empirical analysis. *Journal of Software Maintenance: Research and Practice, 4*(2), 83–104.
6. America, P., & Van, D. L. F., (1990). A parallel object-oriented language with inheritance and subtyping. *ACM SIGPLAN Notices, 25*(10), 161–168.
7. Antolík, J., & Davison, A. P., (2013). Integrated workflows for spiking neuronal network simulations. *Frontiers in Neuroinformatics, 7*, 34.
8. Ashraf, M. U., Fouz, F., & Eassa, F. A., (2016). Empirical analysis of HPC using different programming models. *International Journal of Modern Education & Computer Science, 8*(6), 3–12.
9. Atkinson, M. P., & Buneman, O. P., (1987). Types and persistence in database programming languages. *ACM Computing Surveys (CSUR), 19*(2), 105–170.
10. Bailes, P. A., (1992). Discovering functional programming through imperative languages. *Computer Science Education, 3*(2), 87–110.
11. Bajre, P., & Khan, A., (2019). Developmental dyslexia in Hindi readers: Is consistent sound-symbol mapping an asset in reading? Evidence from phonological and visuospatial working memory. *Dyslexia, 25*(4), 390–410.
12. Berger, U., (2002). Computability and totality in domains. *Mathematical Structures in Computer Science, 12*(3), 281–294.

13. Berry, G., & Gonthier, G., (1992). The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming, 19*(2), 87–152.
14. Blackwell, A. F., Whitley, K. N., Good, J., & Petre, M., (2001). Cognitive factors in programming with diagrams. *Artificial Intelligence Review, 15*(1), 95–114.
15. Blanchet, B., (1999). Escape analysis for object-oriented languages: Application to Java. *ACM SIGPLAN Notices, 34*(10), 20–34.
16. Borning, A., (1981). The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems (TOPLAS), 3*(4), 353–387.
17. Burgess, C. J., & Saidi, M., (1996). The automatic generation of test cases for optimizing Fortran compilers. *Information and Software Technology, 38*(2), 111–119.
18. Burnett, M. M., & Baker, M. J., (1994). A classification system for visual programming languages. *Journal of Visual Languages and Computing, 5*(3), 287–300.
19. Cann, D., (1992). Retire Fortran? a debate rekindled. *Communications of the ACM, 35*(8), 81–89.
20. Cordy, J. R., (2004). TXL-a language for programming language tools and applications. *Electronic Notes in Theoretical Computer Science, 110*, 3–31.
21. Davidsen, M. K., & Krogstie, J., (2010). A longitudinal study of development and maintenance. *Information and Software Technology, 52*(7), 707–719.
22. Davison, A. P., Brüderle, D., Eppler, J. M., Kremkow, J., Muller, E., Pecevski, D., & Yger, P., (2009). PyNN: A common interface for neuronal network simulators. *Frontiers in Neuroinformatics, 2*, 11.
23. Davison, A. P., Hines, M., & Muller, E., (2009). Trends in programming languages for neuroscience simulations. *Frontiers in Neuroscience, 3*, 36.
24. Denning, P. J., (1978). Operating systems principles for data flow networks. *Computer, 11*(07), 86–96.
25. Dubinsky, E., (1995). ISETL: A programming language for learning mathematics. *Communications on Pure and Applied Mathematics, 48*(9), 1027–1051.

26. Egli, H., & Constable, R. L., (1976). Computability concepts for programming language semantics. *Theoretical Computer Science, 2*(2), 133–145.
27. Ferber, J., (1989). Computational reflection in class-based object-oriented languages. *ACM SIGPLAN Notices, 24*(10), 317–326.
28. Fisher, D. A., (1978). DoD's common programming language effort. *Computer, 11*(3), 24–33.
29. Flatt, M., Findler, R. B., Krishnamurthi, S., & Felleisen, M., (1999). Programming languages as operating systems (or revenge of the son of the lisp machine). *ACM SIGPLAN Notices, 34*(9), 138–147.
30. Fourment, M., & Gillings, M. R., (2008). A comparison of common programming languages used in bioinformatics. *BMC Bioinformatics, 9*(1), 1–9.
31. Ghannad, P., Lee, Y. C., Dimyadi, J., & Solihin, W., (2019). Automated BIM data validation integrating open-standard schema with visual programming language. *Advanced Engineering Informatics, 40*, 14–28.
32. Gilmore, D. J., & Green, T. R. G., (1984). Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies, 21*(1), 31–48.
33. Ginsburg, S., & Rose, G. F., (1963). Some recursively unsolvable problems in ALGOL-like languages. *Journal of the ACM (JACM), 10*(1), 29–47.
34. Green, T. R. G., & Petre, M., (1996). Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages & Computing, 7*(2), 131–174.
35. Grove, D., DeFouw, G., Dean, J., & Chambers, C., (1997). Call graph construction in object-oriented languages. In: *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vol. 3, pp. 108–124).
36. Hansen, P. B., (1975). The programming language concurrent pascal. *IEEE Transactions on Software Engineering*, (2), 199–207.
37. Heim, B., Soeken, M., Marshall, S., Granade, C., Roetteler, M., Geller, A., & Svore, K., (2020). Quantum programming languages. *Nature Reviews Physics, 2*(12), 709–722.
38. Heller, J., & Logemann, G. W., (1966). PL/I: A programming language for humanities research. *Computers and the Humanities, 2*, 19–27.

39. Hermenegildo, M. V., Bueno, F., Carro, M., López, P., Morales, J. F., & Puebla, G., (2008). An overview of the ciao multiparadigm language and program development environment and its design philosophy. *Concurrency, Graphs, and Models, 2*, 209–237.
40. Hjelle, K. L., Halvorsen, L. S., & Overland, A., (2010). Heathland development and relationship between humans and environment along the coast of western Norway through time. *Quaternary International, 220*(1, 2), 133–146.
41. Holgeid, K. K., Krogstie, J., & Sjøberg, D. I., (2000). A study of development and maintenance in Norway: Assessing the efficiency of information systems support using functional maintenance. *Information and Software Technology, 42*(10), 687–700.
42. Hutcheon, A. D., & Wellings, A. J., (1988). Supporting Ada in a distributed environment. *ACM SIGAda Ada Letters, 8*(7), 113–117.
43. Ishikawa, Y., Hori, A., Sato, M., Matsuda, M., Nolte, J., Tezuka, H., & Kubota, K., (1996). Design and implementation of metalevel architecture in C++-MPC++ approach. In: *Proceedings of Reflection* (Vol. 96, pp. 153–166).
44. Japaridze, G., (2003). Introduction to computability logic. *Annals of Pure and Applied Logic, 123*(1–3), 1–99.
45. Jeffery, C., Thomas, P., Gaikaiwari, S., & Goettsche, J., (2016). Integrating regular expressions and SNOBOL patterns into string scanning: A unifying approach. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (Vol. 3, pp. 1974–1979).
46. Jones, N. D., (2004). Transformation by interpreter specialization. *Science of Computer Programming, 52*(1–3), 307–339.
47. Kaijanaho, A. J., (2014). The extent of empirical evidence that could inform evidence-based design of programming languages: A systematic mapping study. *Jyväskylä Licentiate Theses in Computing, 2*(18), 1–13.
48. Kari, L., & Thierrin, G., (1996). Contextual insertions/deletions and computability. *Information and Computation, 131*(1), 47–61.
49. Kennedy, K., & Schwartz, J., (1975). An introduction to the set theoretical language SETL. *Computers & Mathematics with Applications, 1*(1), 97–119.
50. King, K. N., (1992). The evolution of the programming languages course. In: *Proceedings of the Twenty-Third SIGCSE Technical Symposium on Computer Science Education* (pp. 213–219).

51. Kiper, J. D., Howard, E., & Ames, C., (1997). Criteria for evaluation of visual programming languages. *Journal of Visual Languages & Computing, 8*(2), 175–192.
52. Krogstie, J., (1996). Use of methods and CASE-tools in Norway: Results from a survey. *Automated Software Engineering, 3*(3), 347–367.
53. Krogstie, J., Jahr, A., & Sjøberg, D. I., (2006). A longitudinal study of development and maintenance in Norway: Report from the 2003 investigation. *Information and Software Technology, 48*(11), 993–1005.
54. Kumar, D., & Wyatt, R., (1995). Undergraduate AI and its non-imperative prerequisite. *ACM SIGART Bulletin, 6*(2), 11–13.
55. Lorenzen, T., (1981). The case for in class programming tests. *ACM SIGCSE Bulletin, 13*(3), 35–37.
56. McCusker, G., (2003). On the semantics of the bad-variable constructor in Algol-like languages. *Electronic Notes in Theoretical Computer Science, 83*, 169–186.
57. McIntyre, D. B., (1991). Language as an intellectual tool: From hieroglyphics to APL. *IBM Systems Journal, 30*(4), 554–581.
58. Mészárosová, E., (2015). Is python an appropriate programming language for teaching programming in secondary schools. *International Journal of Information and Communication Technologies in Education, 4*(2), 5–14.
59. Moot, R., & Retoré, C., (2019). Natural language semantics and computability. *Journal of Logic, Language, and Information, 28*(2), 287–307.
60. Murphree, E. L., & Fenves, S. J., (1970). A technique for generating interpretive translators for problem-oriented languages. *BIT Numerical Mathematics, 10*(3), 310–323.
61. Ottenstein, K. J., Ballance, R. A., & MacCabe, A. B., (1990). The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In: *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, 2*(1), 257–271.
62. Palumbo, D. B., (1990). Programming language/problem-solving research: A review of relevant issues. *Review of Educational Research, 60*(1), 65–89.

63. Patel, P., Torppa, M., Aro, M., Richardson, U., & Lyytinen, H., (2022). Assessing the effectiveness of a game-based phonics intervention for first and second grade English language learners in India: A randomized controlled trial. *Journal of Computer Assisted Learning, 38*(1), 76–89.
64. Persson, M., Bohlin, J., & Eklund, P., (2000). Development and maintenance of guideline-based decision support for pharmacological treatment of hypertension. *Computer Methods and Programs in Biomedicine, 61*(3), 209–219.
65. Raihany, A., & Rabbianty, E. N., (2021). Pragmatic politeness of the imperative speech used by the elementary school language teachers. *OKARA: Journal Bahasa dan Sastra, 15*(1), 181–198.
66. Rajaraman, V., (2014). JohnMcCarthy—Father of artificial intelligence. *Resonance, 19*(3), 198–207.
67. Rao, C., TA, S., Midha, R., Oberoi, G., Kar, B., Khan, M., & Singh, N. C., (2021). Development and standardization of the DALI-DAB (dyslexia assessment for languages of India – dyslexia assessment battery). *Annals of Dyslexia, 71*(3), 439–457.
68. Reddy, U. S., (2002). Objects and classes in Algol-like languages. *Information and Computation, 172*(1), 63–97.
69. Rosen, S., (1971). Programming languages: History and fundamentals (Jean E. Sammet). *SIAM Review, 13*(1), 108.
70. Sammet, J. E., (1972). Programming languages: History and future. *Communications of the ACM, 15*(7), 601–610.
71. Sammet, J. E., (1991). Some approaches to, and illustrations of, programming language history. *Annals of the History of Computing, 13*(1), 33–50.
72. Sanner, M. F., (1999). Python: A programming language for software integration and development. *J Mol Graph Model, 17*(1), 57–61.
73. Smith, D. C., Cypher, A., & Spohrer, J., (1994). KidSim: Programming agents without a programming language. *Communications of the ACM, 37*(7), 54–67.
74. Snyder, A., (1986). Encapsulation and inheritance in object-oriented programming languages. In: *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications* (Vol. 2, No. 2, pp. 38–45).

75. Swanson, E. B., & Beath, C. M., (1990). Departmentalization in software development and maintenance. *Communications of the ACM, 33*(6), 658–667.

76. Sward, R. E., Carlisle, M. C., Fagin, B. S., & Gibson, D. S., (2003). The case for Ada at the USAF academy. In: *Proceedings of the 2003 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems Using Ada and Related Technologies* (Vol. 3, No. 2, pp. 68–70).

77. Swift, J. A., & Mize, J. H., (1995). Out-of-control pattern recognition and analysis for quality control charts using lisp-based systems. *Computers & Industrial Engineering, 28*(1), 81–91.

78. Trichina, E., (1999). Didactic instructional tool for topics in computer science. *ACM SIGCSE Bulletin, 31*(3), 95–98.

79. Valverde, S., & Solé, R. V., (2015). Punctuated equilibrium in the large-scale evolution of programming languages. *Journal of The Royal Society Interface, 12*(107), 20150249.

80. Van, W. A., Mailloux, B. J., Peck, J. E., Kostcr, C. H. A., Sintzoff, M., Lindsey, C. H., & Fisker, R. G., (1977). Revised report on the algorithmic language ALGOL 68. *ACM SIGPLAN Notices, 12*(5), 1–70.

81. Vella, M., Cannon, R. C., Crook, S., Davison, A. P., Ganapathy, G., Robinson, H. P., & Gleeson, P., (2014). libNeuroML and PyLEMS: Using python to combine procedural and declarative modeling approaches in computational neuroscience. *Frontiers in Neuroinformatics, 8*, 38.

82. Vinueza-Morales, M., Borrego, D., Galindo, J. A., & Benavides, D., (2020). Empirical evidence of the usage of programming languages in the educational process. *IEEE Transactions on Education, 64*(3), 213–222.

83. Wasserman, A. I., & Prenner, C. J., (1979). Toward a unified view of database management, programming languages, and operating systems—A tutorial. *Information Systems, 4*(2), 119–126.

84. Yang, H., Torp-Smith, N., & Birkedal, L., (2006). Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *Logical Methods in Computer Science, 2,* 1–23.

CHAPTER 4

INTRODUCTION TO PYTHON PROGRAMMING

CONTENTS

4.1. INTRODUCTION

Computers, in their innate machine language, can understand 0s and 1s. All your computer's executable programs are made up of these 1s and 0s that inform your computer simply what to do. Humans, on the other hand, are terrible at communicating with 0s and 1s (Van Rossum, 2003). Things would go extremely slowly if we had to write our instructions to computers in this fashion all the time, and we would have a lot of disgruntled computer programmers, to say the least (Zhang, 2015). Fortunately, there are two typical ways that programmers may use to avoid having to write their instructions in 0s and 1s to a computer:

- Compiled languages; and
- Interpreted languages.

Compiled languages allow programmers to build programs in a programming language that is easily understandable by humans (Ekmekci et al., 2016). An executable file is created by converting this program into a series of zeros and ones, which is known as an executable file, which the computer can read and comprehend (Liang, 2013; Fangohr, 2015). It is through this executable file that the computer can function. If one wants to make changes to the way their program operates, they must first make the necessary modifications to the program and then recompile (retranslate) the program in order to produce an updated executable file that the computer can recognize and use (Figure 4.1) (Nosrati, 2011; Linge and Langtangen, 2020).

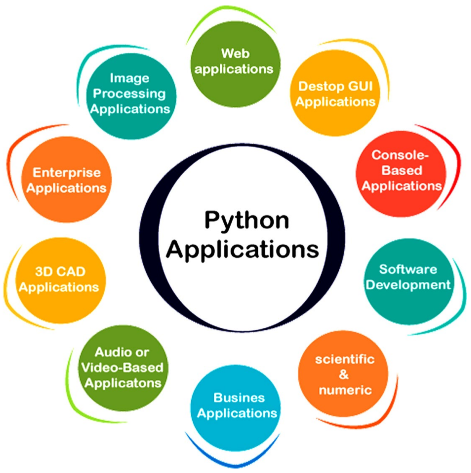
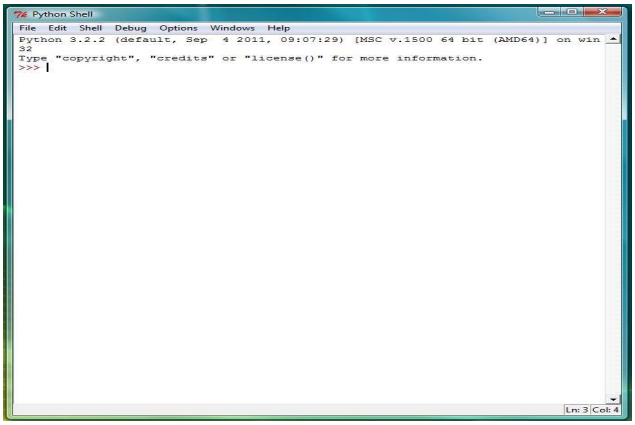


Figure 4.1. Applications of Python programming.

Source: https://www.javatpoint.com/python-applications.

An interpreted language differs from a traditional language in that, instead of performing all the translation rapidly, the compiler first converts a few of this code written in a human-understandable language to an unstructured format, and then this form is "interpreted" into a sequence of 1s and 0s that the machine comprehends and can instantly execute. As a result, both translation and execution are taking place at the same time (Bogdanchikov et al., 2013).

Python is a programming language that is interpreted. IDLE is a common Python programming environment in which students frequently create python applications. It provides students with two distinct ways to build and run Python applications in this environment (Pajankar, 2017). Because Python is an interpreted language, students have the option of writing a single line of python code and seeing the outcomes right away. On the other hand, students can go to a different window, enter all their controls in this window first, and then run their program to see how it functions. The first way allows students to view the outcomes of their assertions in real-time while they are being processed. The second method is more conventional in that it involves first constructing a whole program before compiling it and viewing its results of it. The first strategy is beneficial when it comes to learning. Students, on the other hand, must eventually construct their programs using the second way as a starting point (Van Rossum, 2007; Lakshminarayanan and Prabhakaran, 2020). Figure 4.2 will be presented to you when you open IDLE (Version 3.2.2) for the first time.



```
Python 3.2.2 (default, Sep  4 2011, 09:07:29) [MSC v.1500 64 bit (AMD64)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> 
```

Figure 4.2. Python shell window.

Source: https://zbook.org/read/2752b_-python-chapter-1-introduction-to-programming-in-python-.html.

4.2. OUTPUT: PRINT STATEMENT

4.2.1. Print Statement: Basic Form

The prompt (>>>) asks the user to type a python line that will be interpreted. A print statement is the simplest basic line of code for displaying a result (Figure 4.3) (Taori and Dasararaju, 2019). Consider the following scenario:

>>> print("Hello World!")

The following output will appear when you hit enter from the IDLE editor:

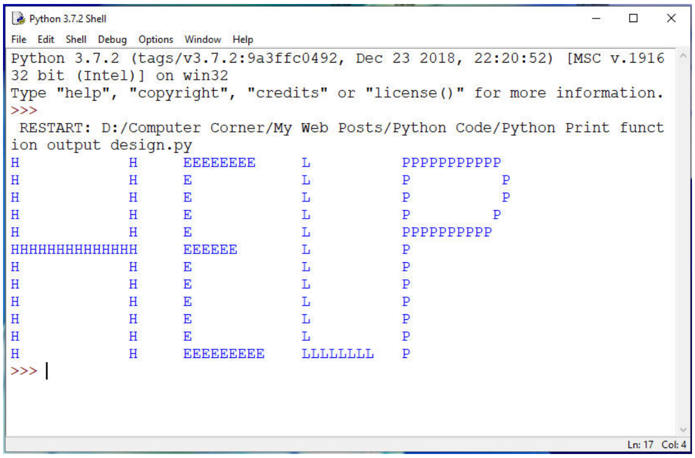
Hello World!

Just look at the following:

>>> print(Hello World)

The following error will appear as IDLE response:

SyntaxError: invalid syntax



```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916
32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: D:/Computer Corner/My Web Posts/Python Code/Python Print funct
ion output design.py
H            H     EEEEEEEE     L          PPPPPPPPPPP
H            H     E            L          P          P
H            H     E            L          P          P
H            H     E            L          P         P
H            H     E            L          PPPPPPPPPP
HHHHHHHHHHHHHH     EEEEEE       L          P
H            H     E            L          P
H            H     E            L          P
H            H     E            L          P
H            H     E            L          P
H            H     E            L          P
H            H     EEEEEEEEE    LLLLLLLL   P
>>> |
```

Figure 4.3. Python print window.

Source: https://computercorner.ca/python-print-function/.

The syntax rules of programming languages are quite rigorous. The difference between programming languages and English is that even if a grammatical rule is violated, most people yet grasp the substance of the message; however, in programming languages, if the tiniest rule is shattered, the interpreter cannot offset by repairing the error (Radenski, 2006; Kadiyala, and Kumar, 2017). Instead, the interpreter generates an error message that informs the programmer of the problem that has occurred. As a result, the message itself is not very useful in this situation because it is not very detailed. In certain circumstances, the error messages are more precise than they are here. When comparing the two statements, it is easy to see that the only thing that separates them is the absence of a pair of double quotes in the second one (which worked). This is the syntax error that was committed previously (Kadiyala and Kumar, 2018).

Having established the correct syntax of the print statement in Python, we can now present it informally:

print('string expression') is a function that prints a string expression.

The term "print" is used first, followed by a pair of enclosing parentheses to complete the sentence (). It is necessary to supply a proper string expression within the parentheses.

The string literal is the first class of string expression we will study. For the purposes of this definition, "literal" means "according to or involving, or consisting of in, or consisting of the fundamental or exact meaning of the word; neither metaphorical nor figurative." Literal simply refers to the concept of "constant" in programming (Srinath, 2017). A literal expression is one that does not have the ability to modify its value. String literals in Python, along with many other programming languages, are denoted by a pair of double quotes that match exactly. With a few exceptions, everything included within the double quotes is regarded as a series of characters, or a string, in the exact same manner as it was written (Tanganelli et al., 2015; Kadiyala and Kumar, 2018).

Consequently, the importance of print ("Hello World!") in Python is just to print out precisely what is included within the double quotes of that phrase. First, try printing out numerous texts that you have written yourself before continuing.

4.2.2. Print Statement: Escape Sequences

You may discover certain restrictions after playing with the print statement. Try printing a message on numerous lines employing a statement with a single print, like the following:

Python is chill!

After entering "Python" in the center of the print statement, one option is to physically hit the enter key. Unfortunately, this results in the mistake:

SyntaxError: EOL as scanning string literal

The term "end of the line" refers to the end of a process. Because the entire Python statements should fit on a single line, the interpreter was waiting to read, represented by the second double quotation, ahead of the close of the line. The interpreter understood the string literal had not been finished when it reached the close of the line, which also signified the close of the statement (Cai et al., 2005; Nagpal and Gabrani, 2019).

To "correct" this problem, we will need a mechanism to tell the translator that we want to go on to the next line without typing the enter key. Python, like several other programming languages, has broken sequences to cope with the problems. A code for a character that should not be interpreted accurately is an avoid sequence (Holkner and Harland, 2009; Saabith et al., 2019). The evade sequence for the latest line character, for example, is n. When these two characters appear in a string literal in such order, the interpreter understands not to display a backslash and an. Instead, it recognizes these two characters as the code for a different line character when they are combined (Manaswi et al., 2018; Kumar and Panda, 2019). Therefore, to print out.

Python is fun!

print("Python\nis\nfun!")

Now there is a list of frequently employed escape sequences:

Character	Escape Sequence
Single-quote	\’
Double quote	\”
Backslash	\\
Tab	\t

The remainder of the information is available in Python's online documentation.

As a result, one method of printing the following is as follows:

Joe says, "Hi!"

is as follows:

print("Joe says, \"Hi!\"")

4.2.3. Second Way to Denote a String Literal in Python

Python varies from other languages in that it offers two methods for specifying string literals. Rather than using double quotes to begin and finish a literal string, single quotes can be used instead. Either option is acceptable (van Rossum, 1995; De Pra et al., 2018). As a result, the above message may be written out more easily as follows:

```
print('Joe says, "Hi!"')
```

The python interpreter understands from the start of the statement that the programmer is employing single quotes to signify the end and start of the string literal, so it may regard the double-quote it finds as a double quote rather than the string's end (Nosrati, 2011).

4.2.4. Automatic Newlines between Prints

When we use IDLE, we are usually obliged to get the outcomes of a single line of code right away. Various genuine computer programs, on the other hand, entail arranging a series of instructions ahead of time and then seeing the outcomes of all those instructions running deprived of having to write in separately novel commands one by one as the program runs (Agarwal and Agarwal, 2006).

We will be able to examine the impact of running two print statements in a row using this method. To do so, just click on the "File" menu in IDLE's main window and pick the first option, "New Window." A single empty window will appear after this option (Agarwal and Agarwal, 2008). Type the subsequent into the window from here:

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

Once you have done so, navigate to the "File" menu in the new tab and select the option "Save As." Select the directory in which you want to store this file and type a name in the box labeled "File Name" in the dialog box that appears (van Rossum and de Boer, 1991). Something simple like hello. py will do the trick. Even though the file type is already displayed below, make sure to include the.py extension. This will make sure that the IDLE editor's emphasis will be visible when you open it. As soon as you have saved the file, you will be able to execute and understand it. Select "Run Module" from the "Run" menu on the main menu bar (Kuhlman, 2009; Kelly, 2019). Following this procedure, you will see the following output:

```
Hello
World!
```

Python, by default, inserts a delimiter character between each print statement, which has resulted in the above situation (Milliken, 2020). While this is typically beneficial, there will be instances in which the programmer does not need the program to necessarily move to the next line of code (Van Rossum et al., 1995; Taori and Dasararaju, 2019). This automated functionality can be turned off by including the following code in the print statement:

print(“Hello,” end = ““) print(“World!”)

With the comma following the literal string, we are informing the print assertion that we have additional evidence for it to consider. To be more specific, we are requesting that our print be terminated with nothing rather than with the standard newline character. Keep in mind that we may put any string in the interior of the double quotes following the equal sign, and anything we identify will be written at the conclusion of the print statement that we are now running. The second print does not follow the same specification as the first, resulting in the newline character being written after the exclamation point in this situation (Watkiss, 2020; Khoirom et al., 2020).

There are various peculiarities to basic printing that need to be considered, but for now, this should be enough. Additional printing regulations will be implemented if needed.

4.2.5. String Operators (+, *)

The Python programming language also has two operators for string chain: string chain (+) and frequent string chain (*). When two strings are concatenated together, it is merely the effect of inserting one string behind another. For instance, the chain of the words “pie” and “apple” results in the phrase “apple pie.” A chain of the similar text more than once is just a function that repeats the similar string a specified number of times. For instance, in Python, multiplying “ahh” by four results in the string “ahhahhahhahh.”

It is important to note that these operators are also valid for numbers and that they are defined in a distinct way for numbers. Overloading is the word used to describe the practice of using two separate definitions for the same thing in a programming language (Van Rossum et al., 2001; Hall and Stacey, 2009). Because of this, the Plus sign in Python is congested and may be used to denote two distinct meanings. (This is a regular occurrence in English.) When it comes to signing anything, the verb “to sign” can indicate either to write one’s signature or to transmit an idea using sign language. By examining the two objects that are being “added,” the computer chooses which of the two meanings to employ (Craven, 2016). Python does string concatenation if both input elements are strings as well. If both variables are numbers, Python adds them together. Python generates an error if one of the items is a string and the other is a number. If, on the other hand, you want to do repeated string concatenation, one of the two objects being multiplied must be a string, and the other must be a non-negative integer. Normal multiplication happens if both items are numbers. If both items are strings, an error occurs if both things are numbers (Hunt, 2019; Bynum et al., 2021). The subsequent instances explain these rules:

```
print(“Happy “+”Anniversary!”) print(4 + 5) print(“4 + 5”) print(“4”+”5”)
print(4*5) print(4*”5”) print(“4”*5) print(“I won’t be available.\n”*3)
```

If we save this segment as a.py file and execute it, we get the following results:

```
Happy  Anniversary!
9
4 + 5
45
20
555
4444
I won’t be available.
I won’t be available.
I won’t be available.
```

The following statements each cause an error:

```
print(4+”5”) print(“4”+5) print(“you”*”me”)
```

The errors are as follows:

```
TypeError: can’t multiply sequence by non-int of type ‘str’
TypeError: unsupported operand type(s) for +: ‘int’ and ‘str’
TypeError: can’t convert ‘int’ object to str implicitly
```

The interpreter alerts you to the fact that a type error has arisen in each circumstance. For the second item, it was anticipating a number in the first statement, a string in the second statement, and another number in the third statement.

4.3. ARITHMETIC EXPRESSIONS: A FIRST LOOK

4.3.1. Standard Operators (+, –, *, /)

Arithmetic computations are one of the most common procedures included in all computer programmers. These are used as parts of entire statements but understanding the principles of arithmetic expressions generally is vital so that we can figure out exactly how the Python interpreter evaluates every expression (Ekmekci et al., 2016). By inputting any arithmetic phrase into the interpreter, we may quickly examine its value:

```
>>> 3+4
7
>>> 17–6
11
>>> 2 + 3*4
14
>>> (2 + 3)*4
20
>>> 3 + 11/4
5.75
```

In a Python application, none of these expressions would ever be used as a full line. The samples provided here are solely for educational purposes. We will learn how to use arithmetic expressions in Python scripts very soon (Alzahrani et al., 2018; Schäfer, 2021).

The four operators are given, multiplication (*), subtraction (–), and addition (+), all function in the same way they did in elementary school. Division and multiplication take precedence over subtraction and addition, as seen in the examples above, and parenthesis can be used to specify the order in which operations should be performed (Dubois et al., 1996; Tateosian, 2015).

4.4. VARIABLES IN PYTHON

4.4.1. The Idea of a Variable

One of the reasons computers programmers are so effective is that they can do computations with a variety of numbers while still following a similar set of instructions. The usage of variables is one method of accomplishing this. Instead of computing 5*5, if we could compute side*side for any value of side, we would be able to measure the area of any square rather than the area of a square with side 5. Variables are simple to utilize in Python. You may insert the name of the variable in your code whenever you wish to utilize it (Donat, 2014; Hunt, 2019). The one drawback is that when you initially make a variable, it lacks a well-defined value, so you cannot utilize it in a framework that requires one.

The simplest approach to introduce a variable is to use an assignment statement, as seen below:

```
>>> side = 5
```

The variable produced is called side, and the line above assigns the value 5 to it (Figure 4.4). The following is a representation of memory at this moment in time:

$$\text{side}\ \boxed{5}$$

If we go along with this statement:

```
>>> area = side*side
```

In memory, then our picture is as follows:

$$\text{side}\ \boxed{5} \qquad \text{area}\ \boxed{25}$$

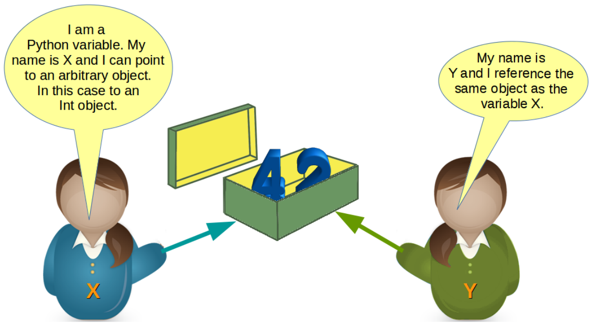

I am a Python variable. My name is X and I can point to an arbitrary object. In this case to an Int object.
My name is Y and I reference the same object as the variable X.
42
X
Y

Figure 4.4. Two Python variables referencing the same object.

Source: *https://python-course.eu/python-tutorial/data-types-and-variables.php.*

Let us look at what is going on. An assignment statement is any statement that has a variable to the left of a single equal sign and an expression to the right of that equal sign (Subero, 2020; Rajagopalan, 2021). An assignment statement's objective is to assign a value to a variable. It operates in a two-step procedure:

- To use the current values of the variables, evaluate the present value of the expression on the right;
- Alter the value of the left-hand variable to this value.

As a result, the side was equal to 5 in the statement above at the time it was performed. As a result, side*side was calculated to be 25. The area box was then replaced with the value 25.

4.4.2. Printing Out the Value of a Variable

Obviously, we do not FIND OUT any indication that the variables are these two values when we run these two lines in IDLE. To do so, we will want to understand that in Python how to print the value of a variable. The most straightforward method is as follows:

```
>>> print(side)
5
>>> print(area)
25
```

It is worth noting that we do not use double quotes in these prints. If we had done the following, we would have:

```
>>> print(“area”) area
>>> print(“side”) side
```

Rather than the values of the relevant variables, the words in concern would have been printed. What we observe here is that anything between double quotes is displayed as is, except for escape sequences, which do not alter. To print the value of a variable, new construction must be used (Hajja et al., 2019; Elumalai, 2021).

“What if we want to publish a variable's value as well as some text in the same print?” is another logical question that occurs. We can achieve this in Python by using commas to separate each thing we want to print, as demonstrated below:

```
>>> print(“The area of a square with side,”side,”is,”area)
```

A square with five sides has a 25-square-foot size.

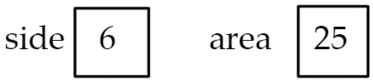
While reading the text above, you will see that Python automatically included a space between each item indicated in print (there are four things total) even though we did not specifically include a space in the output. This is the default configuration in Python, and it is often quite beneficial in many situations (Tang et al., 2014; Nanjekye, 2017). But what if we wanted to put a semicolon directly after the number 25 in the above statement? A space would be added between the number 5 and period if we placed a comma after the area and the string.”“ after it.

4.4.3. Increment Statement

Take into account the statement that comes after the first two statements in the preceding section, which is a little unclear at first:

```
>>> side = side + 1
```

The mathematical concept of a variable equaling itself plus one is known as the equality theorem. This assertion, on the other hand, is not a contradiction in programming. As a result of following the guidelines, we can observe that side is identical at the time of the present assessment. As a result, the right-hand side of the assignment statement is equal to 5 plus 1, which is equal to 6. The next step is to alter the value of the variable on the left to this number, which is 6 (Izaac and Wang, 2018; Pajankar, 2022). The image that corresponds to this sentence is as follows:



Execute the following line to demonstrate that this is really what happened:

```
>>> print(“area =,”area,”side =,”side) area = 6 side = 25
```

One important point to note is that the area is STILL 25. After the side was updated, it did not magically transform to 36. Python only runs the instructions that are passed to it. As a result, if we were to recalculate the area of a square having side 6, we would have to do it.

If we execute the following lines of code again after switching sides, we will get the same result:

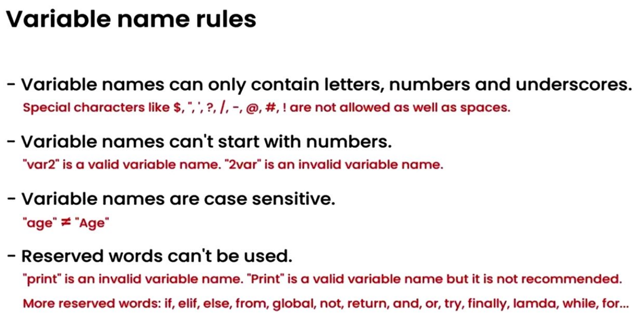
```
>>> area = side*side
>>> print(area)
36
```

Since we expressly reassessed side*side and placed this new value in return into the variable area, we can see that area has now changed to 36.

4.4.4. Rules for Naming a Variable

It goes without saying that a variable cannot be called anything. As an alternative, Python contains criteria for determining which names are applicable for variables and which names are not. To be more specific, the only characters that can be used in a variable name are letters, numerals, and the underscore(“_”). Moreover, the names of variables must not begin with a number (Meulemans et al., 2015; Gerrard, 2016).

Generally, while it is not compulsory, it is regarded excellent programming type to name variables in a way that is related to the function that the variable performs. As seen in the preceding instance, the variables area and side both describe the type of data that has been saved in those variables. If the variables were labeled b and a, for example, somebody else who was reading the code would have a much harder time determining what the function was doing (Rashed et al., 2012; Rawat, 2020). Whether it is out of laziness or for other reasons, many new programmers fall into the practice of designing short variable names that are unrelated to the function of the variable in question. These programmers do not have much trouble with little programmers, but when dealing with bigger programmers, it may be quite difficult to hunt out errors if the role of a variable is not instantly obvious (Figure 4.5) (Oliphant, 2007; Chapman and Stolee, 2016).



Variable name rules

- Variable names can only contain letters, numbers and underscores.
  Special characters like $, ", ', ?, /, -, @, #, ! are not allowed as well as spaces.
- Variable names can't start with numbers.
  "var2" is a valid variable name. "2var" is an invalid variable name.
- Variable names are case sensitive.
  "age" ≠ "Age"
- Reserved words can't be used.
  "print" is an invalid variable name. "Print" is a valid variable name but it is not recommended.
  More reserved words: if, elif, else, from, global, not, return, and, or, try, finally, lamda, while, for...

Figure 4.5. Variable naming rules for Python.

Source: https://www.slideshare.net/p3infotech_solutions/python-programming-essentials-m5-variables.

4.4.5. Two Program Examples

We can create a standalone program using the above set of statements by typing the subsequent in a separate tab and saving it as a python program:

```
# Joe Clark
# 9/10/2019
# The area of a square may be calculated using Python.
side = 5 area = side*side
print(“The area of a square with side,”side,”is,”area)
```

When you run this program, you will get the following results:

```
A square with 5 sides has a 25-square-foot size.
```

4.4.6. Comments

Others find it difficult to read large chunks of code. Programmers frequently include comments in their code to assist others. A comment is a section of code that the interpreter ignores but that anyone viewing the code may see. It provides some fundamental information to the reader. At the start of each program, a header comment is added. It contains information about the file's author(s), the date it was created/edited, and the program's purpose. In Python, the pound sign (#) is used to indicate a comment. The translator treats all text after the pound symbol on a line as a comment (Tateosian, 2015; Poole, 2017).

4.5. ARITHMETIC EXPRESSIONS IN PYTHON

We utilized arithmetic statements on the right-hand side of the assignment declaration in the two examples in the preceding section (equal sign). So that there is no misunderstanding, Python provides its own set of rules for evaluating these expressions. Until now, we have only shown that division and multiplication take preference over subtraction and addition, as is commonly taught in elementary school math (Pilgrim and Willison, 2009; Rak-Amnouykit et al., 2020). Moreover, parentheses have priority over everything else and can also be employed to “force” the order in which operations are assessed, as seen in the previous line of code:

```
total_price = item_price*(1+tax_rate/100)
```

Before multiplying, we analyze the values of the parenthesis in this equation. We do division first when analyzing the content of the parenthesis since it takes priority over addition. For instance, if the tax rate is 7, we divide 7 by 100 to get 0.07, which we then multiply by 1 to obtain 1.07. The present value of item price is then multiplied by 1.07, and the total price is allocated (Meurer et al., 2017; Lukasczyk et al., 2020).

Python additionally gives us with three more operators:

- %, for modulus;
- **, for exponentiation;
- //, for integer division.

Further subsections explain how each of these operators works, as well as the order in which they should be used.

4.5.1. Exponentiation (**)

Because the caret sign () is commonly associated with multiplications on best calculators that children employ in grade school, many pupils learn to associate it with exponentiation initially. Mostly programming languages, however, the caret symbol is either not specified or denotes rather other than involution (Bergstra et al., 2010; Hamrick et al., 2013).

Exponentiation is not defined by an operator in some computer languages, although it is in Python. The operator is only intended to be used with real numbers. Here are a few examples of how it may be used:

```
>>> 2 ** 3
8
>>> 3 ** 5
243
>>> 4 ** 10
1048576
>>> 25 **.5
5.0
>>> 2 ** –3
0.125
>>> 9999 ** 0
1
>>> –5 ** –3
```

```
–0.008
>>> –5 ** 3
–125
>>> 1.6743 ** 2.3233
3.311554089370817
```

When both operands of a multiplications operation are integers, and the result is also an integer, the result is stated as one. The response will be written as a real number with decimals if both operands are integers; however, the answer is not. If an exponent b is –ve, ab is described as 1/a^{-b}, as shown in the instances above (Vanhoenacker and Sandra, 2006; Furduescu, 2019).

4.5.2. Integer Division (/ /)

/ is a second division operator in Python that does integer division. The result of an integer division operation; in particular, is always an integer. The highest number of whole times b divides into an is defined as a/b in particular. Here are a few instances of integer division being used to evaluate expressions:

```
>>> 25//4
6
>>> 13//6
2
>>> 100//4
25
>>> 99//100
0
>>> –17//18
–1
>>> –1//10000000
–1
>>> –99999999//99999999
–1
>>> –25//4 –7
>>> –13//6 –3
>>> –12//6 –2
>>> –11//6
–2
>>> 0//5
0
```

Please keep in mind that Python approaches this operation in a different way than several other programming languages and in a different way than most people's instinctive understanding of integer division. When the majority of people see –13/6, they are likely to conclude that this is quite near to –2, and hence that the answer should be –2, which is incorrect. In contrast, if we look at the technical definition of integer division in Python, we can see that –2 is more than –13/6, which is about –2.166667, and that the highest integer less than or equal to this figure is –3.166667 (Gálvez et al., 2009; Munier et al., 2019).

In addition, the definition of integer division in Python does not need that the two numbers that are being divided be integers in order for the division to take place. Consequently, integer division procedures are permitted even for values that are not in the integer range. Think the following examples:

```
>>> 6.0 // 3.0
2.0
>>> 2.4 // 2.5 0.0
>>> 6.6 //.02
330.0
>>> –45.3 // –11.2
4.0
>>> 45.3 // –11.2
–5.0
```

Because this Python feature is infrequently utilized, no more information will be provided at this time.

4.5.3. Modulus Operator (%)

The modulus operator, which is indicated by the percent sign (%), is likely to be unfamiliar to individuals who have never coded before. When it comes to mathematics, the modulus is normally identified just for integers. In Python, on the other hand, the modulus operator is provided for both real numbers and integers, as opposed to other programming languages. It is possible that the result will be an integer if both integers are operands; otherwise, the answer will be a real number (Henry et al., 1984; Reas and Fry, 2006).

To put it another way: logically, the modulus operator determines the residual in a division, whereas integer division calculates the proportion in a division. Another way of thinking about modulus is that it is just the amount of remaining when two integers are divided by each other. Positive integers in Python are represented correctly by the intuitionistic notion given above. Negative numbers, on the other hand, are a different matter (Bielak, 1993; Nakhle and Harfouche, 2021).

The formal description of a % b is as observes:

a % b assesses to a – (a // b)*b.

The full number of times b splits into an is represented by a / b. As a result, we are searching for the total number of times b enters a and subtracting that many multiples of b from a to get the "leftover."

Here are several traditional mod examples that only use non-negative values:

```
>>> 17 % 3
2
>>> 37 % 4
1
>>> 17 % 9
8
>>> 0 % 17
0
>>> 98 % 7
0
>>> 199 % 200
199
```

In these cases, we can see that if the first value is ever smaller than the second value, the operation's answer is just the first value because the second value divides it 0 times. In the remaining cases, we can see that we can get the solution by simply subtracting the proper number of multiples of the second number (Iyengar et al., 2011; Kopec, 2014).

Negative integers, on the other hand, must be plugged into the formal definition rather than relying on instinct. Consider the following examples:

```
>>> 25 % –6
–5
>>> –25 % –6
–1
>>> –25 % 6
5
>>> –48 % 8
0
>>> –48 % –6
0
>>> –47 % 6
1
>>> –47 % 8
1
>>> –47 % –8
–7
```

The essential problem that determines the first two outcomes is that the two integer divisions 25/–6 and –25/–6 have different answers. The first yields a score of –5, whereas the second yields a score of 4. As a result, we compute –6 × –5 = 30, from which we remove 5 to get 25. For the second, we multiply –6 × 4 to get – 24, then remove 1 to get –25.

Examine if you can use the description provided to understand each of the other responses listed directly above.

The modulus operator is also provided for real numbers in Python, with a similar definition as before. Here are some instances of its use:

```
>>> 12.4 % 6.1
0.2000000000000018
>>> 3.4 % 3.5 3.4
>>> 6.6 %.02
7.216449660063518e–16
>>> –45.3 % –11.2
```

```
–0.5
>>> 45.3 % –11.2
–10.7
```

Looking at the third and first cases, we can see that the result provided by Python is somewhat different from what we would have expected. The first is expected to be just 0.2, and the third is expected to be zero. Sadly, a large quantity of real numbers is not correctly kept in the computer (Jackowska-Strumiłło et al., 2013; Jun Zhao et al., 2013). In fact, this is true in all computer programming languages. Because of this, there are some minor round-off mistakes in computations using real numbers every now and again. Round-off errors are represented by the numbers 1 and 8 at the very end of the number, on the right-hand side of the number. It is worth noting that the last component of the third example is simply the number 10–16 multiplied by the previously revealed number; therefore, the entire section written indicates the round-off error, which is still negligible since it is < 10–15. If we do not require extreme accuracy in our calculations, we may live with the little inaccuracies created by real number computations performed on a standard computer. The more sophisticated the calculations are, the higher the possibility of a mistake cascading to other computations (Chapman and Chang, 2000). However, for the sake of this discussion, we will just believe that our real number of solutions is "near sufficient" to our requirements (Kuhlman, 2009; Chen et al., 2019).

4.6. READING USER INPUT IN PYTHON

4.6.1. Input Statement

Python creates reading user input relatively simple. Python, in particular, ensures that the user is always presented with a prompt to add data. Consider the example given below.

```
>>> name = input("How are you?\n")
    How are you?
    Fine
>>> print("Nice to meet you, "name.,"" sep="") Nice to meet you, Joe.
```

By doing so, rather than the print constantly publishing a similar name, it will print the name that the user has typed into the text box. The important thing to remember is that the input speech read in what the user provided, and then the assignment declaration, which had the equal sign, allocated this value to the variable name. After that, we were free to use any name we wanted, aware that the value given by the user would be saved (Liang, 2013; Derezińska and Hałas, 2014).

It was difficult to expand upon our prior algorithms, which computed the price of an item with tax and the surface area of a square, so they constantly computed the same price and area. The user's input would make our software far more effective if we enabled them to enter the necessary numbers so that our program could assess the information, THEY were keen on (Xu et al., 2021).

Before we get into the specific changes that must be done to these programmers for them to accept user input, we should briefly discuss the input function. It always returns a string representation of whatever data it has received from the user. If the user submits an invalid number, like "79," the input statement will return "79" because of the invalid number. If you want to be literal, this is a string that is the letter "7" subsequently the character "9," rather than the number "79." As a result, we require a technique for converting the text "79" into the number 79. This is accomplished by the use of a function that transforms its input into a new type (Goldbaum et al., 2018; Ortin and Escalada, 2021). The int function can be used to convert a string into an integer:

```
>>> age = int(input("What is your name, "+name+"?\n"))
What is your name? Joe
>>> print("Your name is ",name,". You are ",age," years old.", sep="") Your name is Simone. You are 22 years old.
```

We had to utilize the int function to convert the string returned by the input function keen on an integer in this example. The variable age was then assigned to this. As a result, age saves an integer rather than a string.

You will notice that rather than commas, which we originally used while learning the print statement, plus signs, which denote string concatenation, were used to prompt Simone. This is because, whereas the print statement accepts multiple things separated by commas, the input statement only accepts a single string. As a result, we were obliged to use string concatenation to generate a single string (Ade-Ibijola, 2018; Verstraelen et al., 2021). We were enabled to concatenate the variable name with the remainder of the message since it is a string. If we had struggled to feed the input function distinct items divided by commas, we would have received the following error notice:

```
>>> age = int(input("What is your name, ",name,"?\n"))
```

Traceback (most recent call last): File "<pyshell#13>", line 1, in <module> age = int(input("What is your name, ",name,"?\n")) TypeError: input estimated at most 1 arguments, got 3

The final line of IDLE's yield explains what happened. The input function assumes one parameter, or piece of information, however we supplied it three, because commas are used to divide bits of information (arguments) passed to it.

4.7. EXAMPLES OF PROGRAMS USING THE INPUT() STATEMENT

4.7.1. Making More General Programs

Our examples have all involved writing programmers that performed extremely precise computations, and we have only been able to do it in one case. Consider the case in which we only discovered the area of a single unique square or the price of a single item with tax. Because not all the products we purchase will be the same price, this is not useful (Hedges et al., 2019; Sundnes, 2020).

When it comes to pricing questions, it would be wonderful if the same application could answer them all. This is where we can benefit from user feedback. Rather than being compelled to assign a variable to a certain value, we may merely ask the user to input a number and then set a variable to that value, enabling the user to determine the computation that takes place in the program (Zhu et al., 2018). Here is a version of the software that estimates the area of a square that has been modified to accept user feedback:

```
# Joe Clark
# 9/10/2019
# Python Program to determine the area of a square – using user input.
side = int(input("Please put the side of your square.\n")) area = side*side
print("The area of a square with side,"side,"is,"area)
```

4.7.2. Temperature Conversion Example

Though most people in the United States measure temperature using the Fahrenheit scale, many other people utilize the Celsius system. When a user enters a temperature of Celsius, our application will automatically convert the temperature to Fahrenheit. As an example, consider the following formula for conversion:

$F = 1.8C + 32$

Here is the program:

```
# Joe Clark
# 9/10/2019
# Convert Celsius to Fahrenheit with this program.
temp_cel = float(input("Put the temperature in Celsius.\n")) temp_fahr = 1.8*temp_cel + 32;
print(temp_cel,"degrees Celsius =,"temp_fahr,"degrees Fahrenheit.")
```

Here is an example of the software in action:

```
>>>
Put the temperature in Celsius.
37
37.0 degrees Celsius = 98.60000000000001 degrees Fahrenheit.
>>>
```

4.7.3. Fuel Efficiency Example

Consider the subsequent problem:

You have decided to go on a road trip. The odometer reading on your car's dashboard is visible when you fill up your petrol tank. Later in the journey, you can check how much petrol is left in the tank as well as the mileage on the odometer (O'Boyle et al., 2008; Zandbergen, 2013). We want to figure out how many miles we can travel before we have to stop for petrol again based on all this information. To account for the possibility of making a mistake, we would want to arrive at our destination several kilometers before our gasoline would run out. However, for the sake of simplicity, we will simply compute when we anticipate running out of petrol in this program if we maintain a constant fuel efficiency while driving the car (Robitaille et al., 2013; Li and García, 2021).

This challenge is a little more difficult to solve than the preceding problems. It is preferable to step back and think about the issue, sketching out what variables we want to utilize and how we want to solve it rather than instantly entering the IDLE window.

When we look at the issue statement, we can see that we need to get the subsequent information from the user:

- Initial odometer reading;
- Gas tank size;
- At the halfway point, the odometer reads;
- How much is gas left at the intersection?

The difference between variables 3 and 1 reflects the distance traveled, whereas the difference between variables 2 and 4 shows the quantity of gas consumed during the time period under consideration. The result of dividing the former by the latter will be our fuel economy, expressed in miles per gallon. Because we know how several gallons of petrol is left in the tank, we can multiply that figure by our fuel efficiency to determine how much longer we can continue driving on it (Mukha and Liefvendahl, 2018).

4.8. MATH CLASS

In designing computer programs, some variables and functions that are often connected with mathematics are valuable. These are part of the math library in Python. Many supplementary libraries containing functions to aid the programmer are widespread in Python and practically all other programming languages (Roberts et al., 2010; Sasso et al., 2021). In order to utilize a library in Python, an import statement must be included at the top of the file. We just added the code to import the math library:

import math

At the start of our python file.

Let us have a look at some of the functions and constants in the Python math library:

math.pi is a rough approximation of the circumference to diameter ratio of a circle. math.e – an estimate of the natural logarithm's base.

math.ceil(x) – Returns the smallest integer > or equal to it, as a float. math. fabs(x) – Returns the absolute value of x. math.factorial(x) – Returns x factorial, which is 1 * 2 * 3 *… *x. math.floor(x) – Returns the greatest integer less than or equal to x as a float.

math.exp(x) – Returns e^x. math.log(x) – Returns ln x. math.log(x, base) – Returns $\log_{base}$x. math.sqrt(x) – Returns the square root of x.

math.cos(x) – Returns the cosine of x radians. math.sin(x) – Returns the sine of x radians. math.tan(x) – Returns the tangent of x radians.

To call these functions, we must put "math." previously the name of every function, as seen in the list above. To avoid confusion with other functions having the same name, we need to identify which library the function comes from (Krause and Lindemann, 2014; Guo et al., 2020).

Let us have a look at a handful of Python scripts that use the math package.

4.8.1. Circle Area and Circumference Example

The area of a circle ($A = \pi r^2$) is a common formula given to all geometry students. We will ask the user to enter a circle's radius and print out the associated circumference ($C = 2\pi r$) and area in this software. In our computations, we will utilize the value of pi from the math library.

($C = 2\pi r$)

```
# Joe Clark
# 9/10/2019
# Estimates the circumference and area of a circle, given its radius.
import math
radius = int(input("What is the radius of your circle?\n"))
area = math.pi*(radius**2) circumference = 2*math.pi*radius
print("The area of your circle is ",area,"."",sep=""")
print("The circumference of your circle is ",circumference,"."",sep=""")
```

The running of this program is given:

```
>>>
What is the radius of your circle?
5
The area of your circle is 78.53981633974483.
The circumference of your circle is 31.41592653589793.
>>>
```

4.8.2. Koolaid Example Revisited

The most challenging aspect of the Koolaid program was accurately estimating the number of cups we needed to sell in order to fulfill our profit target. With the following line of code, we avoided an "off by one mistake:

```
num_cups = (target + profit_per_cup – 1) // profit_per_cup
```

If you think about it, what we actually wanted to do was divide the variable's aim and profit per cup on a regular basis, but we only wanted to take the maximum value! For example, if we were required to sell 7.25 cups to meet our objective, we would be unable to do it (Gorgolewski et al., 2011; Langtangen, 2016). We just need to sell the entire cup to make a total of 8 cups! Now that we know how to use the math library, we can update this line of code to make it much more understandable, as seen below:

```
num_cups = int(math.ceil(target/profit_per_cup))
```

4.8.3. Number of Possible Meals

Several restaurants boast about how many different meal options they have. Usually, this just means that they can select a specific quantity of products from a larger batch for the "meal." The number of means to pick k things out of n is, which is interpreted as "n choose k" in mathematics. The formula for calculating a mixture is as follows: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

Let us imagine that a combo meal includes a number of appetizer options out of a total. We will ask the user to input the four values below:

- The maximum number of appetizers that can be served;
- The number of appetizers that can be included in the combination is limited;
- Total possible number of entrees; and
- The combo's maximum number of entrees.

In order to compute both of the necessary combinations, we may make use of the factorial function. In order to achieve our final answer, we can merely add the results of both calculations together to get our final answer. This is because each conceivable option of appetizers can be matched with each possible choice of entrees (VanderPlas et al., 2018). For example, a two-dimensional framework with the rows labeled by all appetizers and the columns labeled by all entrees may be used to represent this.

In this instance, we will make one last change to our program before we are finished. For several programming languages, code begins by performing from a function called the main function. While it is not required in Python, it is a good practice to get into the habit of identifying a function main in any programming language. It will be beneficial when transferring to other programming languages, and as the python programs, you develop become longer than a few lines, having the main function will be handy from an organizational standpoint. This may be included in your program by simply including the subsequent line just before your program's guidelines:

```
def main():
```

Python necessitates constant indenting; therefore, every statement within the main function must be indented as well. Four spaces or a tab are used to indicate a normal indentation. Following the completion of your code in main, you must invoke the function mainly because all you have done so far has been to declare the function (Combrisson et al., 2017). However, just because a function is defined does not imply that it will be used. It is only utilized if and when it is requested. The following is an example of how to invoke the function main:

```
main()
```

Putting this all together, we have the subsequent program:

```
# Joe Clark
# 9/10/2019
# Analyzes the number of possible combo meals. import math
# Several languages define a function main, which starts execution. def main():
# Get the user information. numapps = int(input("How many total appetizers are there?\n")) yourapps = int(input("How many of those do you get to choose?\n")) numentrees = int(input("How many total entrees are there?\n")) yourentrees = int(input("How many of those do you get to choose?\n"))
#Calculate the combinations of appetizers and entrees.
appcombos = (math.factorial(numapps)/math.factorial(yourapps)
/math.factorial(numapps-yourapps)) entreecombos = (math.factorial(numentrees)/math.factorial(yourentrees) /math.factorial(numentrees-yourentrees))
# Output the final answer.
print("You can order," int(appcombos*entreecombos), "different meals.")
#Call main!
main()
```

A single line of code that spans two lines is another new feature in this application. This happens with both the app combos and entree combos' assignment statements. An extra set of parentheses is needed to persuade Python to understand that the whole expression goes on a single line. There are various ways to signal that many lines of code correspond to a single line of code, but this is the recommended one:

```
appcombos = math.factorial(numapps)/math.factorial(yourapps) \
/math.factorial(numapps-yourapps)
```

REFERENCES

1. Agarwal, K. K., & Agarwal, A., (2006). Simply python for CS 0. *Journal of Computing Sciences in Colleges, 21*(4), 162–170.
2. Agarwal, K., Agarwal, A., & Celebi, M. E., (2008). Python puts a squeeze on java for CS0 and beyond. *Journal of Computing Sciences in Colleges, 23*(6), 49–57.
3. Alzahrani, N., Vahid, F., Edgcomb, A., Nguyen, K., & Lysecky, R., (2018). Python versus C++ an analysis of student struggle on small coding exercises in introductory programming courses. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (pp. 86–91).
4. Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., & Bengio, Y., (2010). Theano: A CPU and GPU math compiler in python. In: *Proc. 9th Python in Science Conf.* (Vol. 1, pp. 3–10).
5. Bielak, R., (1993). Object oriented programming: The fundamentals. *ACM SIGPLAN Notices, 28*(9), 13–14.
6. Bogdanchikov, A., Zhaparov, M., & Suliyev, R., (2013). Python to learn programming. In: *Journal of Physics: Conference Series* (Vol. 423, No. 1, p. 012027). IOP Publishing.
7. Bynum, M. L., Hackebeil, G. A., Hart, W. E., Laird, C. D., Nicholson, B. L., Siirola, J. D., & Woodruff, D. L., (2021). A brief python tutorial. In: *Pyomo—Optimization Modeling in Python* (pp. 203–216). Springer, Cham.
8. Cai, X., Langtangen, H. P., & Moe, H., (2005). On the performance of the python programming language for serial and parallel scientific computations. *Scientific Programming, 13*(1), 31–56.
9. Chapman, B., & Chang, J., (2000). Biopython: Python tools for computational biology. *ACM SIGBIO Newsletter, 20*(2), 15–19.
10. Chapman, C., & Stolee, K. T., (2016). Exploring regular expression usage and context in python. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis* (pp. 282–293).
11. Chen, T., Hague, M., Lin, A. W., Rümmer, P., & Wu, Z., (2019). Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proceedings of the ACM on Programming Languages, 3*(POPL), 1–30.

12. Combrisson, E., Vallat, R., Eichenlaub, J. B., O'Reilly, C., Lajnef, T., Guillot, A., & Jerbi, K., (2017). Sleep: An open-source python software for visualization, analysis, and staging of sleep data. *Frontiers in Neuroinformatics, 11*, 60.
13. Craven, P. V., (2016). Create a custom calculator. In: *Program Arcade Games* (pp. 11–31). A press, Berkeley, CA.
14. De Pra, Y., Fontana, F., & Simonato, M., (2018). Development of real-time audio applications using python. In: *Proceedings of the XXII Colloquium of Musical Informatics, Udine, Italy* (pp. 22–23).
15. De-Ibijola, A., (2018). Syntactic generation of practice novice programs in python. In: *Annual Conference of the Southern African Computer Lecturers' Association* (pp. 158–172). Springer, Cham.
16. Derezińska, A., & Hałas, K., (2014). Analysis of mutation operators for the python language. In: *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX* (pp. 155–164). Brunów, Poland. Springer, Cham.
17. Donat, W., (2014). Introducing python. In: *Learn Raspberry Pi Programming with Python* (pp. 31–50). A press, Berkeley, CA.
18. Dubois, P. F., Hinsen, K., & Hugunin, J., (1996). Numerical python. *Computers in Physics, 10*(3), 262–267.
19. Ekmekci, B., McAnany, C. E., & Mura, C., (2016). An introduction to programming for bioscientists: A python-based primer. *PLoS Computational Biology, 12*(6), e1004867.
20. Elumalai, A., (2021). Python loves numbers. In: *Introduction to Python for Kids* (pp. 39–58). Apress, Berkeley, CA.
21. Fangohr, H., (2015). Introduction to python for computational science and engineering. *Faculty of Engineering and the Environment University of Southampton*, 68.
22. Furduescu, B. A., (2019). Neuro-linguistic programming: History, conception, fundamentals, and objectives. *Valahian Journal of Economic Studies, 10*(1).
23. Gálvez, J., Guzmán, E., & Conejo, R., (2009). A blended e-learning experience in a course of object oriented programming fundamentals. *Knowledge-Based Systems, 22*(4), 279–286.
24. Gerrard, P., (2016). Input and output. In: *Lean Python* (pp. 35–41). Apress, Berkeley, CA.

25. Goldbaum, N. J., ZuHone, J. A., Turk, M. J., Kowalik, K., & Rosen, A. L., (2018). *unyt: Handle, Manipulate, and Convert Data with Units in Python.* arXiv preprint arXiv:1806.02417.
26. Gorgolewski, K., Burns, C. D., Madison, C., Clark, D., Halchenko, Y. O., Waskom, M. L., & Ghosh, S. S., (2011). Nipype: A flexible, lightweight, and extensible neuroimaging data processing framework in python. *Frontiers in Neuroinformatics, 5*, 13.
27. Guo, P. J., Markel, J. M., & Zhang, X., (2020). Learner sourcing at scale to overcome expert blind spots for introductory programming: A three-year deployment study on the python tutor website. In: *Proceedings of the Seventh ACM Conference on Learning@ Scale* (pp. 301–304).
28. Hajja, A., Hunt, A. J., & McCauley, R., (2019). PolyPy: A web-platform for generating quasi-random python code and gaining insights on student learning. In: *2019 IEEE Frontiers in Education Conference (FIE)* (pp. 1–8). IEEE.
29. Hall, T., & Stacey, J. P., (2009). Variables and data types. *Python 3 for Absolute Beginners,* 27–47.
30. Hamrick, T. R., & Hensel, R. A., (2013). Putting the fun in programming fundamentals-robots make programs tangible. In: *2013 ASEE Annual Conference & Exposition* (pp. 23–1012).
31. Hedges, L. O., Mey, A. S., Laughton, C. A., Gervasio, F. L., Mulholland, A. J., Woods, C. J., & Michel, J., (2019). BioSimSpace: An interoperable python framework for biomolecular simulation. *Journal of Open Source Software, 4*(43), 1831.
32. Henry, R. C., Lewis, C. W., Hopke, P. K., & Williamson, H. J., (1984). Review of receptor model fundamentals. *Atmospheric Environment (1967), 18*(8), 1507–1515.
33. Holkner, A., & Harland, J., (2009). Evaluating the dynamic behavior of python applications. In: *Proceedings of the Thirty-Second Australasian Conference on Computer Science* (Vol. 91, pp. 19–28).
34. Hunt, J., (2019). A first python program. In: *A Beginners Guide to Python 3 Programming* (pp. 23–31). Springer, Cham.
35. Hunt, J., (2019). Python modules and packages. In: *A Beginners Guide to Python 3 Programming* (pp. 281–297). Springer, Cham.
36. Iyengar, S. S., Parameshwaran, N., Phoha, V. V., Balakrishnan, N., & Okoye, C. D., (2011). *Fundamentals of Sensor Network Programming: Applications and Technology* (Vol. 41, pp. 1–36). John Wiley & Sons.

37. Izaac, J., & Wang, J., (2018). Python. In: *Computational Quantum Mechanics* (pp. 83–162). Springer, Cham.
38. Jackowska-Strumiłło, L., Nowakowski, J., Strumiłło, P., & Tomczak, P., (2013). Interactive question based learning methodology and clickers: Fundamentals of computer science course case study. In: *2013 6th International Conference on Human System Interactions (HSI)* (pp. 439–442). IEEE.
39. Jun, Z. Y., Ying, Z. C., & Wang, J., (2013). Innovative practices teaching mode research of the fundamentals of computer. In: *2013 8th International Conference on Computer Science & Education* (pp. 1154–1159). IEEE.
40. Kadiyala, A., & Kumar, A., (2017). Applications of python to evaluate environmental data science problems. *Environmental Progress & Sustainable Energy, 36*(6), 1580–1586.
41. Kadiyala, A., & Kumar, A., (2018). Applications of python to evaluate the performance of decision tree-based boosting algorithms. *Environmental Progress & Sustainable Energy, 37*(2), 618–623.
42. Kadiyala, A., & Kumar, A., (2018). Applications of python to evaluate the performance of bagging methods. *Environmental Progress & Sustainable Energy, 37*(5), 1555–1559.
43. Kelly, S., (2019). Introducing python. In: *Python, PyGame, and Raspberry Pi Game Development* (pp. 11–31). Apress, Berkeley, CA.
44. Khoirom, S., Sonia, M., Laikhuram, B., Laishram, J., & Singh, T. D., (2020). Comparative analysis of python and Java for beginners. *Int. Res. J. Eng. Technol., 7*(8), 4384–4407.
45. Kopec, D., (2014). Some programming fundamentals. In: *Dart for Absolute Beginners* (pp. 15–24). Apress, Berkeley, CA.
46. Krause, F., & Lindemann, O., (2014). Expyriment: A python library for cognitive and neuroscientific experiments. *Behavior Research Methods, 46*(2), 416–428.
47. Kuhlman, D., (2009). *A Python Book: Beginning Python, Advanced Python, and Python Exercises* (pp. 1–227). Lutz: Dave Kuhlman.
48. Kumar, A., & Panda, S. P., (2019). A survey: How python pitches in it-world. In: *2019 International Conference on Machine Learning, Big Data, Cloud, and Parallel Computing (COMITCon)* (pp. 248–251). IEEE.

49. Lakshminarayanan, D., & Prabhakaran, S., (2020). A study on python programming language. *Dogo Rangsang Res. J., 10*, 2347–7180.

50. Lamy, J. B., (2017). Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies. *Artificial Intelligence in Medicine, 80*, 11–28.

51. Langtangen, H. P., (2016). User input and error handling. In: *A Primer on Scientific Programming with Python* (pp. 149–225). Springer, Berlin, Heidelberg.

52. Li, Z., & García, M. H., (2021). pyRiverBed: A python framework to generate synthetic riverbed topography for constant-width meandering rivers. *Computers & Geosciences, 152*, 104755.

53. Liang, Y. D., (2013). For introduction to programming using python. *Displays, 8*(8), 8.

54. Linge, S., & Langtangen, H. P., (2020). *Programming for Computations-Python: A Gentle Introduction to Numerical Simulations with Python 3.6* (p. 332). Springer Nature.

55. Lukasczyk, S., Kroiß, F., & Fraser, G., (2020). Automated unit test generation for python. In: *International Symposium on Search-Based Software Engineering* (pp. 9–24). Springer, Cham.

56. Manaswi, N. K., Manaswi, N. K., & John, S., (2018). *Deep Learning with Applications Using Python* (pp. 31–43). Bangalore, India: Apress.

57. Meulemans, J., Ward, T., & Knights, D., (2015). Syntax and semantics of coding in python. In: *Hydrocarbon and Lipid Microbiology Protocols* (pp. 135–154). Springer, Berlin, Heidelberg.

58. Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., & Scopatz, A., (2017). SymPy: Symbolic computing in python. *Peer J. Computer Science, 3*, e103.

59. Milliken, C. P., (2020). Python basics. In: *Python Projects for Beginners* (pp. 21–46). Apress, Berkeley, CA.

60. Mukha, T., & Liefvendahl, M., (2018). Eddylicious: A python package for turbulent inflow generation. *SoftwareX*, *7*, 112–114.

61. Munier, N., Hontoria, E., & Jiménez-Sáez, F., (2019). Linear programming fundamentals. In: *Strategic Approach in Multi-Criteria Decision Making* (pp. 101–116). Springer, Cham.

62. Nagpal, A., & Gabrani, G., (2019). Python for data analytics, scientific, and technical applications. In: *2019 Amity International Conference on Artificial Intelligence (AICAI)* (pp. 140–145). IEEE.

63. Nakhle, F., & Harfouche, A. L., (2021). Ready, steady, Go AI: A practical tutorial on fundamentals of artificial intelligence and its applications in phenomics image analysis. *Patterns, 2*(9), 100323.

64. Nanjekye, J., (2017). Printing and backtick repr. In: *Python 2 and 3 Compatibility* (pp. 1–10). Apress, Berkeley, CA.

65. Nosrati, M., (2011). Python: An appropriate language for real world programming. *World Applied Programming, 1*(2), 110–117.

66. O'Boyle, N. M., Morley, C., & Hutchison, G. R., (2008). Pybel: A python wrapper for the OpenBabel cheminformatics toolkit. *Chemistry Central Journal, 2*(1), 1–7.

67. Oliphant, T. E., (2007). Python for scientific computing. *Computing in Science & Engineering, 9*(3), 10–20.

68. Ortin, F., & Escalada, J., (2021). Cnerator: A python application for the controlled stochastic generation of standard C source code. *SoftwareX, 15*, 100711.

69. Pajankar, A., (2017). Introduction to python. In: *Python Unit Test Automation* (pp. 1–17). Apress, Berkeley, CA.

70. Pajankar, A., (2022). Introduction to python 3. In: *Hands-on Matplotlib* (pp. 1–28). Apress, Berkeley, CA.

71. Pilgrim, M., & Willison, S., (2009). *Dive into Python 3* (Vol. 2, pp. 20–30). New York, NY, USA: Apress.

72. Poole, M., (2017). Extending the design of a blocks-based python environment to support complex types. In: *2017 IEEE Blocks and Beyond Workshop (B&B)* (pp. 1–7). IEEE.

73. Radenski, A., (2006). " Python first" a lab-based digital introduction to computer science. *ACM SIGCSE Bulletin, 38*(3), 197–201.

74. Rajagopalan, G., (2021). Getting familiar with python. In: *A Python Data Analyst's Toolkit* (pp. 1–43). Apress, Berkeley, CA.

75. Rak-amnouykit, I., McCrevan, D., Milanova, A., Hirzel, M., & Dolby, J., (2020). Python 3 types in the wild: A tale of two type systems. In: *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages* (pp. 57–70).

76. Rashed, M. G., & Ahsan, R., (2012). Python in computational science: Applications and possibilities. *International Journal of Computer Applications, 46*(20), 26–30.

77. Rawat, A., (2020). A review on python programming. *International Journal of Research in Engineering, Science, and Management, 3*(12), 8–11.
78. Reas, C., & Fry, B., (2006). Processing: Programming for the media arts. *Ai & Society, 20*(4), 526–538.
79. Roberts, J. J., Best, B. D., Dunn, D. C., Treml, E. A., & Halpin, P. N., (2010). Marine geospatial ecology tools: An integrated framework for ecological geoprocessing with ArcGIS, python, R, MATLAB, and C++. *Environmental Modeling & Software, 25*(10), 1197–1207.
80. Robitaille, T. P., Tollerud, E. J., Greenfield, P., Droettboom, M., Bray, E., Aldcroft, T., & Streicher, O., (2013). Astropy: A community python package for astronomy. *Astronomy & Astrophysics, 558*, A33.
81. Saabith, A. S., Fareez, M. M. M., & Vinothraj, T., (2019). Python current trend applications-an overview. *International Journal of Advance Engineering and Research Development, 6*(10).
82. Sasso, A., Morgenstern, J., Musmann, F., & Arnrich, B., (2021). Devicely: A python package for reading, time shifting and writing sensor data. *Journal of Open Source Software, 6*(66), 3679.
83. Schäfer, C., (2021). The basic structure of a python program. In: *Quickstart Python* (pp. 9–11). Springer, Wiesbaden.
84. Srinath, K. R., (2017). Python: The fastest growing programming language. *International Research Journal of Engineering and Technology (IRJET), 4*(12), 354–357.
85. Subero, A., (2021). Python programming. In: *Programming Microcontrollers with Python* (pp. 107–125). Apress, Berkeley, CA.
86. Sundnes, J., (2020). User input and error handling. In: *Introduction to Scientific Programming with Python* (pp. 57–80). Springer, Cham.
87. Tang, T., Rixner, S., & Warren, J., (2014). An environment for learning interactive programming. In: *Proceedings of the 45th ACM technical symposium on Computer Science Education* (pp. 671–676).
88. Tanganelli, G., Vallati, C., & Mingozzi, E., (2015). CoAPthon: Easy development of CoAP-based IoT applications with python. In: *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)* (pp. 63–68). IEEE.
89. Taori, P., & Dasararaju, H. K., (2019). Introduction to python. In: *Essentials of Business Analytics* (pp. 917–944). Springer, Cham.

90. Tateosian, L., (2015). Beginning python. In: *Python for ArcGIS* (pp. 13–35). Springer, Cham.
91. Tateosian, L., (2015). *Python for ArcGIS* (p. 544). Cham, Switzerland: Springer.
92. van Rossum, G., & de Boer, J., (1991). Interactively testing remote servers using the python programming language. *CWi Quarterly, 4*(4), 283–303.
93. Van, R. G., & Drake, Jr. F. L., (1995). *Python Tutorial* (Vol. 620, pp. 250–290). Amsterdam, The Netherlands: Centrum Voor Wiskunde en Informatica.
94. Van, R. G., (2003). In: Drake, F. L., (ed.), *An Introduction to Python* (p. 115). Bristol: Network Theory Ltd.
95. Van, R. G., (2007). Python programming language. In: *USENIX Annual Technical Conference* (Vol. 41, No. 1, pp. 1–36).
96. Van, R. G., Warsaw, B., & Coghlan, N., (2001). PEP 8-style guide for python code. *Python. Org., 1565.*
97. VanderPlas, J., Granger, B. E., Heer, J., Moritz, D., Wongsuphasawat, K., Satyanarayan, A., & Sievert, S., (2018). Altair: Interactive statistical visualizations for python. *Journal of Open Source Software, 3*(32), 1057.
98. Vanhoenacker, G., & Sandra, P., (2006). Elevated temperature and temperature programming in conventional liquid chromatography: Fundamentals and applications. *Journal of separation Science, 29*(12), 1822–1835.
99. vanRossum, G., (1995). Python reference manual. *Department of Computer Science [CS]*, (R 9525).
100. Verstraelen, T., Adams, W., Pujal, L., Tehrani, A., Kelly, B. D., Macaya, L., & Heidar-Zadeh, F., (2021). IOData: A python library for reading, writing, and converting computational chemistry file formats and generating input files. *Journal of Computational Chemistry, 42*(6), 458–464.
101. Watkiss, S., (2020). Getting started with python. In: *Beginning Game Programming with Pygame Zero* (pp. 11–49). Apress, Berkeley, CA.
102. Xu, D., Liu, B., Feng, W., Ming, J., Zheng, Q., Li, J., & Yu, Q., (2021). Boosting SMT solver performance on mixed-bitwise-arithmetic expressions. In: *Proceedings of the 42nd ACM SIGPLAN International*

Conference on Programming Language Design and Implementation (pp. 651–664).

103. Zandbergen, P. A., (2013). *Python Scripting for ArcGIS* (p. 358). Redlands, CA: Esri press.
104. Zhang, Y., (2015). An introduction to python and computer programming. In: *An Introduction to Python and Computer Programming* (pp. 1–11). Springer, Singapore.
105. Zhu, M., McKenna, F., & Scott, M. H., (2018). OpenSeesPy: Python library for the OpenSees finite element framework. *SoftwareX, 7*, 6–11.

CHAPTER 5

FUNDAMENTALS OF C PROGRAMMING

CONTENTS

5.1. INTRODUCTION

C is a common programming language that may be used to create programs for a wide range of purposes, including operating systems (OSs), numerical computation, and graphical applications. With just 32 keywords, it is a little language. It supports both "high-level" structured programming tools like looping, decision making, and statement grouping, along with "low-level" capabilities like manipulating addresses and bytes (Embree et al., 1991; Rajon, 2016).

Because C is a tiny language, it can be explained in a short amount of time and learned rapidly. A programmer may fairly expect to know, comprehend, and utilize the complete language on a regular basis (Figure 5.1) (Mészárosová, 2015).

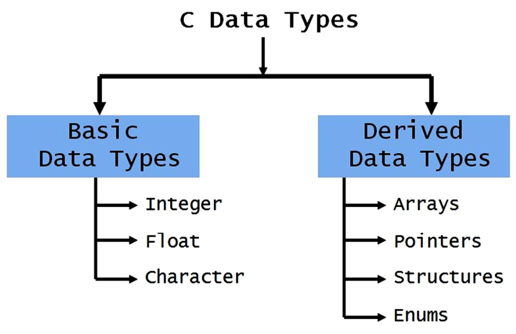


Figure 5.1. Fundamentals of C data types.

Source: https://talentcode.blogspot.com/2020/04/fundamentals-of-c-programming.html.

As a result, C is able to maintain its small size by offering just the most basic functions inside the language itself and by omitting several of the higher-level elements that are often found in other languages. In contrast to other programming languages, C does not include any operations that interact with composite objects such as arrays or lists. Aside from the static declaration of local variables and the stack-allocation of those variables, there are no memory management features. In addition, there are no input/output capabilities, like writing to a file or printing to the screen on the computer (Vogel-Heuser et al., 2014).

A large portion of C functionality is provided by software routines known as functions. An extensive standard library of functions is provided with the language to facilitate the execution of routinely performed activities. Take, for instance, the standard function printf(), which outputs text to the screen (or more properly, to standard output, which is usually the screen). In this work, the standard library will be utilized extensively; thus, it is crucial to avoid developing your own code when a suitable and accessible implementation already exists in the standard library (Oliveira et al., 2013; Qian and Lau, 2017).

5.2. A FIRST PROGRAM

A C program, no matter how big or little, is made up of variables and functions. Variables hold values utilized during the computation, and statements indicate the computational processes to be performed.

The program that follows is the standard first program taught in beginning C courses and textbooks (Figure 5.2).

```c
/* First C program: Hello World */
#include <stdio.h>

int main(void)
{
        printf("Hello World!\n");
}
```

Figure 5.2. The first program in C.

Source: https://freecomputerbooks.com/C-Programming-Language-and-Software-Design.html.

Comments in C begin with a /* and end with a */. They are not nestable and can span numerous lines. For instance,

/* this makes an effort to nest two comments /* results in just one comment, ending here: */ and the residual text is a syntax error. */

A typical library header file is included. Libraries provide the majority of C's functionality. Header files include information such as function definitions and macros that are required to utilize these libraries.

The entry-point function for all C programs is main(). There are two types of this function:

int main(void)

int main(int argc, char *argv[])

The first accepts no parameters, whereas the second takes command-line inputs from the program's execution environment—typically a command shell. The function returns an integer value (i.e., an integer).

The brackets define the function block's boundaries. When a function is finished, the program returns to the function that is called it. The program stops when main() is called, and control passes to the environment in which the program was run. The program's exit status to the environment is indicated by the integer return value of main(), with 0 indicating regular termination (Mardan, 2014).

This program just has one statement: a call to the printf() function in the standard library, which outputs a character string to standard output. Note that printf() is a function supplied by the standard library, not a part of the C language. The typical library is a set of functions that must be present on all ISO C-compliant computers. The printf() method accepts only one argument in this case: the string constant "Hello World!" The n at the end of the string is an escape character that indicates the beginning of a new line. Escape characters are used to indicate characters that are difficult to type or are not visible. Ultimately, a semicolon marks the end of the sentence (;). In most cases, C is a free-form language, with program meaning intact by whitespace. As a result, statements are finished rather than by a new line (Chan et al., 1992; Kenner et al., 2010).

5.3. VARIANTS OF HELLO WORLD

The output from the subsequent program is equal to that produced by the previous example. A further line is not automatically created with every call to printf(), and succeeding strings are merely adjoined together till the escape character (n) is encountered, as demonstrated (Figure 5.3).

```
/* Hello World version 2 */
#include <stdio.h>

int main(void)
{
        printf("Hello ");
        printf("World!");
        printf("\n");
}
```

Figure 5.3. Hello word version 2.

Source: https://www.getfreeebooks.com/an-introduction-to-the-c-programming-language-and-software-design/.

"Hello, World!" is also printed by the following software. But rather than publishing the entire string at once, it prints each character as it is received. A number of new ideas are introduced as a result of this exercise: identifiers, variables, types, pointers, the 0 (NUL) escape character, array subscripts, increment operators, logical operators, whereas loops, and text formatting, among others (Backus, 2003).

This may appear to be a lot, but do not be concerned; you are not required to grasp everything right once, and everything will be discussed in greater detail in the following chapters. At this point, it is sufficient to comprehend the fundamental structure of the code: an index argument, a loop, a string, and a print statement (Figure 5.4) (McMillan, 2018).

```
/* Hello World version 3 */
#include <stdio.h>

int main(void)
{
        int i = 0;
        char *str = "Hello World!\n";

        /* Print each character until reach '\0' */
        while (str[i] != '\0')
                printf("%c", str[i++]);

        return 0;
}
```

Figure 5.4. Hello word version 3.

Source: https://freecomputerbooks.com/C-Programming-Language-and-Software-Design.html.

Before they may be utilized, all variables must be defined. They must be defined before any statements at the head of a block. When declared, they can be started by an expression or a constant.

The variable with the identifier I is of the type int, which is an integer with a value of zero. The identifier str refers to a variable of type char *, which is a character pointer. The characters in a string constant are referred to as str in this example.

A while-loop repeats through the string, printing each character one by one. The loop continues to run as long as the expression (str[i]!= '0') is non-zero. NOT EQUAL TO is the meaning of the operator!=. The i-th character in a string is referred to as str[i] (where str[0] is 'H'). The escape letter '0' specifies that all string constants be indirectly ended with a NUL character (Caprile and Tonella, 1999; Dimovski et al., 2021).

While the loop expression is TRUE, the while-loop runs the following sentence. The printf() function accepts two inputs in this case: a format string " percent c" and a constraint str[i++], then outputs the i-th character of str. The post-increment operator is the expression i++, which returns the value of I and then increases it to I = I + 1.

This version of the program, unlike earlier versions, provides an exact return statement indicating the program's exit status.

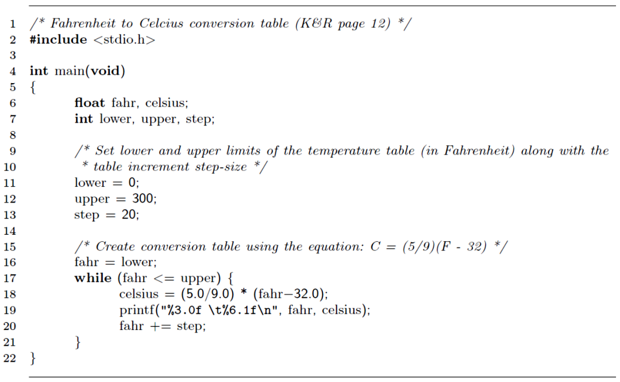
- ***Style Note:*** Take note of the structuring style employed in the sample code throughout this text, especially the indentation. Indentation is an important part of designing readable C programs. Indentation is not important to the compiler, although it does make the program simpler to understand for programmers (Wang et al., 2014; Medeiros et al., 2015).

5.4. A NUMERICAL EXAMPLE

This program makes use of a number of variables. These have to be stated at the beginning of a block, first before statements are written. Variables are defined by their types, which in this case are int and float, respectively.

Please keep in mind that the * at the beginning of line 10 is not essential and is just used for cosmetic purposes.

The three integer variables are initialized in the program by the first three statements in the program (Figure 5.5).



```
1   /* Fahrenheit to Celcius conversion table (K&R page 12) */
2   #include <stdio.h>
3
4   int main(void)
5   {
6           float fahr, celsius;
7           int lower, upper, step;
8
9           /* Set lower and upper limits of the temperature table (in Fahrenheit) along with the
10           * table increment step-size */
11          lower = 0;
12          upper = 300;
13          step = 20;
14
15          /* Create conversion table using the equation: C = (5/9)(F - 32) */
16          fahr = lower;
17          while (fahr <= upper) {
18                  celsius = (5.0/9.0) * (fahr-32.0);
19                  printf("%3.0f \t%6.1f\n", fahr, celsius);
20                  fahr += step;
21          }
22  }
```

Figure 5.5. Fahrenheit to Celsius conversion table.

Source: https://codecondo.com/20-ways-to-learn-c-programming-for-free/c-programming-language-and-software-design-by-tim-bailey/.

In this step, we initialize the floating-point variable fahr. Take note of the fact that the two variables are of a distinct type. For types that are compatible with one another, the compiler conducts automated type conversion (Schilling, 1995, Duff, 2015).

The while-loop is activated whenever the expression (fahr = upper) evaluates to FALSE. The operator = denotes that something is < or = something else. This loop performs a compound statement encased in braces, which corresponds to the three statements on the first and second lines of code (Austin et al., 1994).

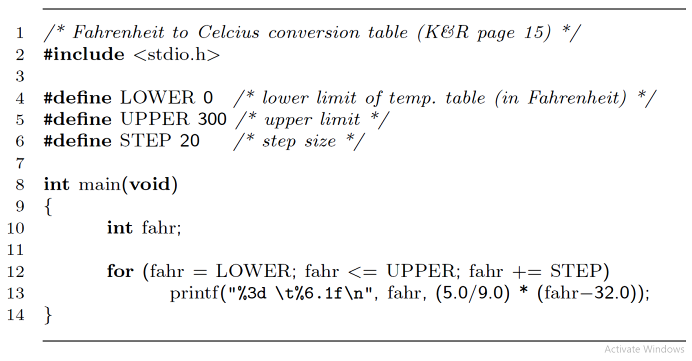
The printf() command, in this case, is made up of two variables and a format string, Celsius, and fahr, that are used to display the results. With two conversion specifiers, percent 3.0f and percent 6.1f, and tab and newline, two escape characters, the format string can be easily read. For example, the conversion specifier percent 6.1f formats a floating-point number by providing space for at least six digits and printing one digit just after the decimal point, and printing one digit after the decimal point (Westerståhl, 1985; Kimura and Tanaka-Ishii, 2014).

+= generates an expression that is equal to the expression fahr = fahr plus step.

- ***Style Note:*** In order to make the code more understandable, comments should be utilized. They should explain the goal of the algorithm and point out intricacies in the method. They should refrain from repeating code slang. It is possible to significantly minimize the number of comments necessary to make understandable code by carefully selecting identifiers.

5.5. ANOTHER VERSION OF THE CONVERSION TABLE EXAMPLE

This version of the exchange table example yields the same results as the first but adds symbolic constants and a for-loop (Figure 5.6).



```
/* Fahrenheit to Celcius conversion table (K&R page 15) */
#include <stdio.h>

#define LOWER 0   /* lower limit of temp. table (in Fahrenheit) */
#define UPPER 300 /* upper limit */
#define STEP 20   /* step size */

int main(void)
{
        int fahr;

        for (fahr = LOWER; fahr <= UPPER; fahr += STEP)
            printf("%3d \t%6.1f\n", fahr, (5.0/9.0) * (fahr−32.0));
}
```

Figure 5.6. Fahrenheit to Celsius conversion table with symbolic constants.

Source: http://www.freebookcenter.net/programming-books-download/An-Introduction-to-the-C-Programming-Language-and-Software-Design-(PDF-158P).html.

Names for numerical constants are known as symbolic constants. These are defined using #define, and they allow us to avoid having numbers pollute our code. Magic numbers are numbers that are strewn about in code and should be avoided at all costs (Feldmann et al., 1998; Ferreira, 2003).

Two semicolons divide the three components of the for-loop (;). The first modifies the loop, the second verify the condition, and the third is an expression that is run after every loop iteration. The real conversion expression is contained within the printf() statement; an expression can be employed everywhere a variable can be used (Gravley and Lakhotia, 1996).

- ***Style Note:*** Multi-word names should be written like this, and variables should always start with a lowercase letter. To distinguish them from variables, symbolic constants should always be written in UPPERCASE.

5.6. IDENTIFIERS

Identifiers (variable names, function names, and so on) are case-sensitive and made up of letters and numbers. An identifier's initial character must be a letter, including underscore ().

The C programming language features 32 reserved keywords that cannot be employed as identifiers (e.g., int, while, etc.). Furthermore, avoiding redefining identifiers employed against the C standard library is a smart idea

(Ambriola et al., 1985; Giannotti et al., 1987).

- ***Style Note:*** For variable names, use lowercase, while for symbolic constants, use uppercase. External variable names should be longer and more informative than local variable names. Variable names can start with an underscore (_). However, this is discouraged because such names are reserved for library implements by convention.

5.7. TYPES

C is a typed programming language. Every variable has a type that specifies what values it may signify, how its data is kept in memory, and what actions it can execute. The type system lets the compiler catch type-disparity issues by compelling the programmer to explicitly declare a type for all variables and interfaces, therefore preventing a large source of faults (Miné, 2006; Majumdar and Xu, 2007).

In the C programming language, there are three main types: characters, integers, and floating-point numbers.

The numerical kinds are available in a variety of sizes. A collection of C types and their generally Works Data Types may be found in Table 5.1.

Table 5.1. C Data Types and Their Normal Sizes

Int	Generally, the natural word size for an OS or machine
Char	Usually 8-bits (1 byte)
Long int	At least 32-bits
Short int	As a minimum of 16-bits
Float	Generally, 32-bits
Long double	Generally, at least 64-bits
Double	Usually 64-bits

Sizes may differ from one platform to the next. Almost every modern processor represents an integer with a minimum of 32 bits, and several increasingly utilize 64 bits. In general, the size of an int indicates a machine's natural word size, the indigenous size with which the CPU processes data and instructions (Lahiri et al., 2012; Irlbeck, 2015).

The standard simply says that a short int must be as a minimum of 16 bits and a long int must be at least 32 bits in size, and short int ≤ int ≤ long int

Except for that, the standard states nothing concerning the size of floating-point numbers.

float ≤ double ≤ long double.

Below is a program that prints the range of values for various data formats. In standard headers limitations, parameters like INT MIN can be found float.h (Figure 5.7).

```
#include <stdio.h>
#include <limits.h> /* integer specifications */
#include <float.h> /* floating-point specifications */

/* Look at range limits of certain types */
int main (void)
{
        printf("Integer range:\t%d\t%d\n", INT_MIN, INT_MAX);
        printf("Long range:\t%ld\t%ld\n", LONG_MIN, LONG_MAX);
        printf("Float range:\t%e\t%e\n", FLT_MIN, FLT_MAX);
        printf("Double range:\t%e\t%e\n", DBL_MIN, DBL_MAX);
        printf("Long double range:\t%e\t%e\n", LDBL_MIN, LDBL_MAX);
        printf("Float-Double epsilon:\t%e\t%e\n", FLT_EPSILON, DBL_EPSILON);
}
```

Figure 5.7. Code for looking at range limits of types.

Source: http://www.freebookcenter.net/programming-books-download/An-Introduction-to-the-C-Programming-Language-and-Software-Design-(PDF-158P).html.

- ***Note:*** The size of the operator can be used to determine the size of a type in characters. This operator is not a function, despite its appearance. It is a keyword. It yields a size t unsigned integer, which is specified in the stddef.h header file (Figure 5.8).

```
#include <stdio.h>

int main (void)
/* Print the size of various types in "number-of-chars" */
{
        printf("void\tchar\tshort\tint\tlong\tfloat\tdouble\n");
        printf("%3d\t%3d\t%3d\t%3d\t%3d\t%3d\t%3d\n",
                sizeof(void), sizeof(char), sizeof(short), sizeof(int),
                sizeof(long), sizeof(float), sizeof(double));
}
```

Figure 5.8. Code for printing size of various types.

Source: https://www.getfreeebooks.com/an-introduction-to-the-c-programming-language-and-software-design/.

Since they change the size of a fundamental int type, the keywords short and long are known as type qualifiers. Note the difference between long and short when employed alone, as in short x.

short a; and long a; are the equivalents of long int and short int. Unsigned, signed, volatile, and const are other types of qualifiers. The qualifiers unsigned and signed can be applied to any integer type, including char. A signed type can hold negative values; the sign-bit is the number's most significant bit (MSB), and the value is usually stored in 2's complement binary. A 16-bit signed short, for example, may signify the integers 32,768 to 32,767, but a 16-bit unsigned short can express the numbers 0 to 65,535 (Ball and Rajamani, 2001; Alturki, 2017).

- ***Note:*** By default, integer types are signed. Plain chars, on the other hand, are either unsigned or signed by default, depending on the computer.

The qualifier const indicates that the variable to which it references is immutable.

const int DoesNotChange = 5;

DoesNotChange = 6; /* Error: will not compile */

The qualifier volatile is used to refer to variables whose value may vary in a way that is outside the power of the program's usual operations. This is important for things like multi-threaded programming or interacting with hardware, which are issues that are outside the range of this document. The volatile qualifier is not appropriate to standard-compliant C programs, and as a result, it will not be discussed in any further detail in this chapter (McMillan, 1993; Yang and Seger, 2003).

Furthermore, there is a type called void, which denotes a type that has "no value" associated with it. In functions that do not take any arguments, it is used as an argument, and in functions that return no value, it is used as a return type.

5.8. CONSTANTS

Different types and presentations of constants exist. This section gives examples of various constant types. First, the type of the integer constant 1234 is int. The suffix L, 1234L, is added to a long int constant. A U, 1234U, denotes an unsigned int, while UL denotes an unsigned long (Bryant et al., 2002; Ringenburg and Grossman, 2005).

In addition to decimal values, integer constants can be given using a hexadecimal or octal values. A 0 precedes octal numerals, and a 0 precedes hex numbers. As a result, 1234 is the decimal equal of 02,322 and 0x4D2. It is vital to keep in mind that these three constants all embody a similar thing (0101 1101 0010 in binary)—for instance, the subsequent code.

```
int x = 1234, y = 02322, z = 0x4D2;
printf("%d\t%o\t%x\n", x, x, x);
printf("%d\t%d\t%d\n", x, y, z);
```

It is worth noting that C lacks a direct binary representation. The hex form, on the other hand, is particularly helpful in reality since it divides binary into four-bit chunks.

After an integer, a decimal point is used to specify floating-point constants. For instance, 1. and 1.3 are double types, 3.14f and 2.f is float types, and 7.L is a long double type. The scientific notation may also be used to write floating-point values, like 1.65e-2. At compilation time, constant expressions like 3+7+9.2 are assessed and swapped with a single constant value, 19.2. Constant expressions have no runtime above as a result (Jayaram and Prasad, 2011; Trudel et al., 2012).

Single quotes are used to provide character constants such as ‘a,’ ‘n,’ and ‘7.’ Character constants are notable since they are of the int type rather than the char type. On a 32-bit computer, size of (‘Z’) will equal four, not one. The ASCII character set correlates the integers 0 to 127 with particular characters.

Certain characters are specified via an “escape sequence” since they cannot be rendered directly. It is crucial to remember that these escape characters are, however, single characters. The following is a list of important escape characters: 0 stands for null, t for tab, n for newline, b for backslash, v for vertical tab, b for backspace, “ for double quotes, and ‘ for single quotes (Shahriar and Zulkernine, 2008; Boudjema et al., 2018).

Quotes are used to separate string constants like “This is a string.” A termination ‘0’ character is implicitly attached to them. As a result, the aforementioned string constant would have the subsequent character sequence in memory: This is a string of zero characters (Brooks, 1999; Vedala and Kumar, 2012).

Ø ***Note:*** It is crucial to understand the difference between a character constant and a NUL ending string constant. The latter is made up of two X0 characters concatenated together. Similarly, size of ('X') on a 32-bit computer is four, but sizeof("X") is two.

5.9. SYMBOLIC CONSTANTS

Symbolic constants are names that characterize constant values from the above-mentioned collection of constant kinds. For instance,

#define TRACK_SIZE	(16*BLOCK_SIZE)
#define BLOCK_SIZE	100
#define HELLO	"Hello World\n"
#define EXP	2.7183

A symbolic constant is comparable to a direct sentence with the constant it specifies everywhere it occurs in the code. printf(HELLO); for example, produces the string Hello World. The use of symbolic constants rather than direct constant values minimizes the spread of "magic numbers"—numerical constants strewn across the code. This is critical since magic numbers are susceptible to errors and provide a significant challenge when trying to make code modifications. Symbolic constants store constants in one location so that changes may be made quickly and safely (Radzi et al., 2016).

- ***Note:*** The #define symbol is a preprocessor command, similar to the #include a symbol for file inclusion. As a result, it follows a distinct set of rules than the fundamental C language. It is important to note that the # must be the first character on a line and not indented.

An Enum, which is a list of constant integer values, is another type of symbolic constant.

For instance, Enum Boolean TRUE, FALSE; The enumeration tag Boolean identifies the enumeration list's "type," allowing a variable of that type to be specified.

Enum Boolean x = FALSE;

If an enumeration list is described, not including an exact tag, it believes the type int. For instance,

Enum { GREEN =2, RED, BLUE, YELLOW=4, BLACK }; int y = BLUE;

By default, the value of an enumeration list begins at zero and increases by one for each successive item. Non-specified members are every > the preceding member, and list members can be provided explicit integer values.

- ***Style Note:*** Uppercase names are provided to symbolic constants and identifiers as a matter of convention. This distinguishes them from variables and functions, which should always start with a lowercase letter, according to the convention. Const-qualified variables operate as constants. Hence, they should be named in uppercase or have the initial letter capitalized (Thomas, 1953; Ventura et al., 2015).

5.10. PRINTF CONVERSION SPECIFIERS

The standard function printf() makes it easy to output formatted text. It uses several formatting operators and conversion specifiers to combine numerical values of any type into a character string (Figure 5.9) (Joseph, 2018).

```
printf("Character values %c %c %c\n", ’a’, ’b’, ’c’);
printf("Some floating-point values %f %f %f\n", 3.556, 2e3, 40.1f);
printf("Scientific notation %e %e %e\n", 3.556, 2e3, 40.1f);
printf("%15.10s\n", "Hello World\n"); /* Right-justify string with space for
                                         15 chars, print only first 10 letters */
```

Figure 5.9. Display (printf) code in C.

Source: https://www.getfreeebooks.com/an-introduction-to-the-c-programming-language-and-software-design/.

- ***Important*:** A conversion specifier and the variable it refers to must be of the same type. If they are not, the software will either crash or output junk. printf(“percent f,” 52); / is an example. * Integer value with floating-point specifier */

REFERENCES

1. Alturki, M. A., (2017). A symbolic rewriting semantics of the COMPASS modeling language. In: *2017 IEEE International Conference on Information Reuse and Integration (IRI)* (pp. 283–290). IEEE.
2. Ambriola, V., Giannotti, F., Pedreschi, D., & Turini, F., (1985). Symbolic semantics and program reduction. *IEEE Transactions on Software Engineering*, (8), 784–794.
3. Austin, T. M., Breach, S. E., & Sohi, G. S., (1994). Efficient detection of all pointer and array access errors. In: *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation* (pp. 290–301).
4. Backus, D. J. (2003). *2.9. 1 Obsolescence and Deletions 2.9. 2" Hello World" Example 2.10 Fortran 95 2.10. 1 Conditional Compilation and Varying Length Strings 2.11 Fortran 2003* (Vol. 3).
5. Ball, T., & Rajamani, S. K., (2001). Automatically validating temporal safety properties of interfaces. In: *International SPIN Workshop on Model Checking of Software* (pp. 102–122). Springer, Berlin, Heidelberg.
6. Boudjema, E. H., Faure, C., Sassolas, M., & Mokdad, L., (2018). Detection of security vulnerabilities in C language applications. *Security and Privacy, 1*(1), e8.
7. Brooks, D. R., (1999). The basics of C programming. In: *C Programming: The Essentials for Engineers and Scientists* (pp. 23–69). Springer, New York, NY.
8. Bryant, R. E., Lahiri, S. K., & Seshia, S. A., (2002). Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: *International Conference on Computer Aided Verification* (pp. 78–92). Springer, Berlin, Heidelberg.
9. Caprile, C., & Tonella, P., (1999). Nomen est omen: Analyzing the language of function identifiers. In: *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)* (pp. 112–122). IEEE.
10. Chan, S. W., McOmish, F., Holmes, E. C., Dow, B., Peutherer, J. F., Follett, E., & Simmonds, P., (1992). Analysis of a new hepatitis C virus type and its phylogenetic relationship to existing variants. *Journal of General Virology, 73*(5), 1131–1141.

11. Dehnert, J. C., & Stepanov, A., (2000). Fundamentals of generic programming. In: *Generic Programming* (pp. 1–11). Springer, Berlin, Heidelberg.
12. Dimovski, A. S., Apel, S., & Legay, A., (2021). Program sketching using lifted analysis for numerical program families. In: *NASA Formal Methods Symposium* (pp. 95–112). Springer, Cham.
13. Duff, M. J., (2015). How fundamental are fundamental constants? *Contemporary Physics, 56*(1), 35–47.
14. Embree, P. M., Kimble, B., & Bartram, J. F., (1991). *C Language Algorithms for Digital Signal Processing* (pp. 15–20).
15. Feldmann, T., Kroll, P., & Stech, B., (1998). Mixing and decay constants of pseudoscalar mesons. *Physical Review D, 58*(11), 114006.
16. Ferreira, C., (2003). Function finding and the creation of numerical constants in gene expression programming. In: *Advances in Soft Computing* (pp. 257–265). Springer, London.
17. Giannotti, F., Matteucci, A., Pedreschi, D., & Turini, F., (1987). Symbolic evaluation with structural recursive symbolic constants. *Science of Computer Programming, 9*(2), 161–177.
18. Gravley, J. M., & Lakhotia, A., (1996). Identifying enumeration types modeled with symbolic constants. In: *Proceedings of WCRE'96: 4th Working Conference on Reverse Engineering* (pp. 227–236). IEEE.
19. Irlbeck, M., (2015). Deconstructing dynamic symbolic execution. *Dependable Software Systems Engineering, 40*, 26.
20. Jayaram, M. A., & Prasad, D. R., (2011). *Programming in C Language* (Vol. 1, pp. 150–180). Sapna Book House (P) Ltd.
21. Joseph, L., (2018). Fundamentals of C++ for robotics programming. In: *Robot Operating System (ROS) for Absolute Beginners* (pp. 55–94). Apress, Berkeley, CA.
22. Kenner, A., Kästner, C., Haase, S., & Leich, T., (2010). Typechef: Toward type checking# ifdef variability in c. In: *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development* (pp. 25–32).
23. Kimura, D., & Tanaka-Ishii, K., (2014). Study on constants of natural language texts. *Information and Media Technologies, 9*(4), 771–789.
24. Lahiri, S. K., Hawblitzel, C., Kawaguchi, M., & Rebêlo, H., (2012). Symdiff: A language-agnostic semantic diff tool for imperative

programs. In: *International Conference on Computer Aided Verification* (pp. 712–717). Springer, Berlin, Heidelberg.

25. Majumdar, R., & Xu, R. G., (2007). Directed test generation using symbolic grammars. In: *Proceedings of the Twenty-Second IEEE/ACM international conference on Automated Software Engineering* (pp. 134–143).
26. Mardan, A., (2014). Hello world example. In: *Pro Express. JS* (pp. 15–30). Apress, Berkeley, CA.
27. McMillan, K. L., (1993). The SMV system. In: *Symbolic Model Checking* (pp. 61–85). Springer, Boston, MA.
28. McMillan, S., (2018). Making containers easier with HPC container maker. In: *Proceedings of the SIGHPC Systems Professionals Workshop (HPCSYSPROS 2018), Dallas, TX, USA* (Vol. 10, pp. 150–180).
29. Medeiros, F., Rodrigues, I., Ribeiro, M., Teixeira, L., & Gheyi, R., (2015). An empirical study on configuration-related issues: Investigating undeclared and unused identifiers. *ACM SIGPLAN Notices, 51*(3), 35–44.
30. Mészárosová, E., (2015). Is python an appropriate programming language for teaching programming in secondary schools. *International Journal of Information and Communication Technologies in Education, 4*(2), 5–14.
31. Miné, A., (2006). Symbolic methods to enhance the precision of numerical abstract domains. In: *International Workshop on Verification, Model Checking, and Abstract Interpretation* (pp. 348–363). Springer, Berlin, Heidelberg.
32. Oliveira, O. L., Monteiro, A. M., & Roman, N. T., (2013). Can natural language be utilized in the learning of programming fundamentals? In: *2013 IEEE Frontiers in Education Conference (FIE)* (pp. 1851–1856). IEEE.
33. Qian, C., & Lau, K. K., (2017). Enumerative variability in software product families. In: *2017 International Conference on Computational Science and Computational Intelligence (CSCI)* (pp. 957–962). IEEE.
34. Radzi, N. A. M., Ismail, A., Karunanithi, S., Weng, L. Y., Jern, K. P., Hock, G. C., & Krishnan, P. S., (2016). Integrating programming with BeagleBone black for undergraduate's "programming for engineers" syllabus. In: *2016 IEEE 8th International Conference on Engineering Education (ICEED)* (pp. 12–15). IEEE.

35. Rajon, S. A., (2016). *Fundamentals of Computer Programming with C* (Vol. 10, pp. 25–35). SA AHSAN RAJON.
36. Ringenburg, M. F., & Grossman, D., (2005). Preventing format-string attacks via automatic and efficient dynamic checking. In: *Proceedings of the 12th ACM conference on Computer and Communications Security* (pp. 354–363).
37. Schilling, J. L., (1995). Dynamically-valued constants: An underused language feature. *ACM SIGPLAN Notices, 30*(4), 13–20.
38. Shahriar, H., & Zulkernine, M., (2008). Mutation-based testing of format string bugs. In: *2008 11th IEEE High Assurance Systems Engineering Symposium* (pp. 229–238). IEEE.
39. Shevlyakov, A. N., (2015). Algebraic geometry over Boolean algebras in the language with constants. *Journal of Mathematical Sciences, 206*(6), 742–757.
40. Thomas, W. H., (1953). Fundamentals of digital computer programming. *Proceedings of the IRE, 41*(10), 1245–1249.
41. Trudel, M., Furia, C. A., Nordio, M., Meyer, B., & Oriol, M., (2012). C to OO translation: Beyond the easy stuff. In: *2012 19th Working Conference on Reverse Engineering* (pp. 19–28). IEEE.
42. Vedala, R., & Kumar, S. A., (2012). Automatic detection of printf format string vulnerabilities in software applications using static analysis. In: *Proceedings of the CUBE International Information Technology Conference* (pp. 379–384).
43. Ventura, M., Ventura, J., Baker, C., Viklund, G., Roth, R., & Broughman, J., (2015). Development of a video game that teaches the fundamentals of computer programming. In: *SoutheastCon 2015* (pp. 1–5). IEEE.
44. Vogel-Heuser, B., Rehberger, S., Frank, T., & Aicher, T., (2014). Quality despite quantity—Teaching large heterogenous classes in C programming and fundamentals in computer science. In: *2014 IEEE Global Engineering Education Conference (EDUCON)* (pp. 367–372). IEEE.
45. Wang, Y., Wang, C., Li, X., Yun, S., & Song, M., (2014). *How are Identifiers Named in Open Source Software? About Popularity and Consistency* (pp. 4–9). *arXiv preprint arXiv:1401.5300.*
46. Westerståhl, D., (1985). Logical constants in quantifier languages. *Linguistics and Philosophy,* 387–413.

47. Yang, J., & Seger, C. J., (2003). Introduction to generalized symbolic trajectory evaluation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 11*(3), 345–353.

CHAPTER 6

DYNAMIC PROGRAMMING

CONTENTS

6.1. INTRODUCTION

Dynamic programming is an optimization approach that transforms a complex problem into a sequence of simpler problems. The essential characteristic of dynamic programming is the multistage nature of the optimization procedure. More so than the optimization techniques described previously, dynamic programming provides a general framework for analyzing many problem types. Within this framework a variety of optimization techniques can be employed to solve particular aspects of a more general formulation. Usually, creativity is required before we can recognize that a particular problem can be cast effectively as a dynamic program; and often subtle insights are necessary to restructure the formulation so that it can be solved effectively (Amini et al., 1990; Osman et al., 2005).

We begin by providing a general insight into the dynamic programming approach by treating a simple example in some detail. We then give a formal characterization of dynamic programming under certainty, followed by an in-depth example dealing with optimal capacity expansion. Other topics covered in the chapter include the discounting of future returns, the relationship between dynamic-programming problems and shortest paths in networks, an example of a continuous-state-space problem, and an introduction to dynamic programming under uncertainty (Powell et al., 2002; Momoh, 2009).

6.2. AN ELEMENTARY EXAMPLE

In order to introduce the dynamic-programming approach to solving multistage problems, in this section we analyze a simple example. Figure 6.1 represents a street map connecting homes and downtown parking lots for a group of commuters in a model city. The arcs correspond to streets and the nodes correspond to intersections (Birge and Louveaux, 2011; Rust, 2008). The network has been designed in a diamond pattern so that every commuter must traverse five streets in driving from home to downtown. The design characteristics and traffic pattern are such that the total time spent by any commuter between intersections is independent of the route taken (Held et al., 1962; Bellman et al., 2015). However, substantial delays, are experienced by the commuters in the intersections. The lengths of these delays in minutes, are indicated by the numbers within the nodes. We would like to minimize the total delay any commuter can incur in the intersections while driving from his home to downtown. Figure 6.2 provides a compact tabular representation for the problem that is convenient for discussing its solution by dynamic programming. In this figure, boxes correspond to intersections in the network. In going from home to downtown, any commuter must move from left to right through this diagram, moving at each stage only to an adjacent box in the next column to the right (Snyder et al., 1987; Ulmer et al., 2019). We will refer to the "stages to go," meaning the number of intersections left to traverse, not counting the intersection that the commuter is currently in.

A naive approach to solving the problem would be to enumerate all 150 paths through the diagram, selecting the path that gives the smallest delay. Dynamic programming reduces the number of computations by moving systematically from one side to the other, building the best solution as it goes (Sali and Blundell, 1990).

Suppose that we move backward through the diagram from right to left. If we are in any intersection (box) with no further intersections to go, we have no decision to make and simply incur the delay corresponding to that intersection (Barto et al., 1995; Huan and Marzouk, 2016). The last column in Figure 6.2 summarizes the delays with no (zero) intersections to go.

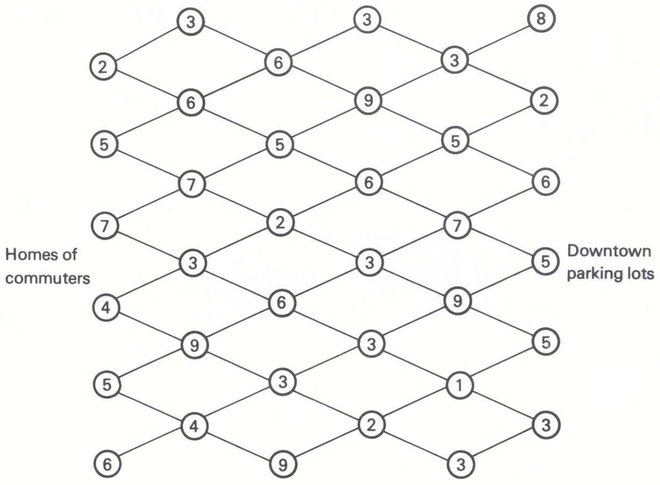


Figure 6.1. Street map with intersection delays.

Source: https://www.researchgate.net/figure/Street-map-with-intersection-delays-Taken-from-30_fig4_330557459.

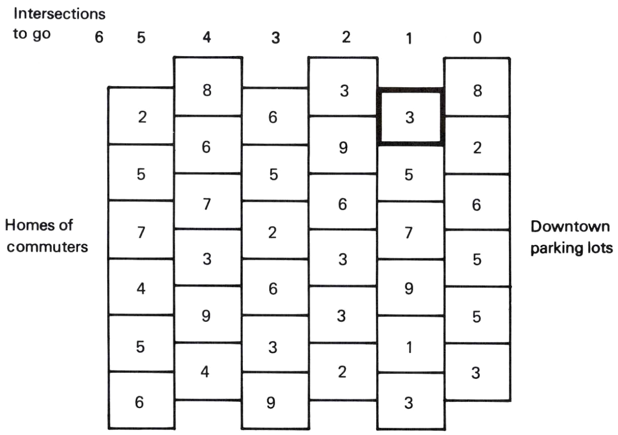


Figure 6.2. Compact representation of the network.

Source: https://www.researchgate.net/publication/330557459_Testing_Time-of-Use_and_Subscription_based_grid_tariff_structures_using_a_Prosumer_model.

Our first decision (from right to left) occurs with one stage, or intersection, left to go. If for example, we are in the intersection corresponding to the highlighted box in Figure 6.2, we incur a delay of three minutes in this intersection and a delay of either *eight* or *two* minutes in the last intersection, depending upon whether we move up or down. Therefore, the smallest possible delay, or optimal solution, in this intersection is 3+2 = 5 minutes (Li et al., 2014; Jamal et al., 2014). Similarly, we can consider each intersection (box) in this column in turn and compute the smallest total delay as a result of being in each intersection. The solution is given by the bold-faced numbers in Figure 6.3. The arrows indicate the optimal decision, up or down, in any intersection with one stage, or one intersection, to go (Sen and Head, 1997; Guo et al., 2019).

Note that the numbers in bold-faced type in Figure 6.3 completely summarize, for decision-making purposes, the total delays over the last two columns. Although the original numbers in the last two columns have been used to determine the bold-faced numbers, whenever we are making decisions to the left of these columns, we need only know the bold-faced numbers. In an intersection, say the topmost with one stage to go, we know that our (optimal) remaining delay, including the delay in this intersection, is five minutes. The bold-faced numbers summarize all delays from this point on. For decision-making to the left of the bold-faced numbers, the last column can be ignored (Yagar and Han, 1994; Kappelman and Sinha, 2021).

With this in mind, let us back up one more column, or stage, and compute the optimal solution in each intersection with two intersections to go (Battigalli and Siniscalchi, 2002; Dayan and Daw, 2008). For example, in the bottom-most intersection, which is highlighted in Figure 6.3, we incur a delay of two minutes in the intersection, plus *four* or *six* additional minutes, depending upon whether we move up or down. To minimize delay, we move *up* and incur a total delay in this intersection and *all remaining intersections* of 2 + 4 = 6 minutes. The remaining computations in this column are summarized in Figure 6.4, where the bold-faced numbers reflect the optimal total delays in each intersection with two stages, or two intersections, to go (Van Damme, 1989; Hauk et al., 2002).

Once we have computed the optimal delays in each intersection with two stages to go, we can again move back one column and determine the optimal delays and the optimal decisions with three intersections to go. In the same way, we can continue to move back one stage at a time, and compute the optimal delays and decisions with four and five intersections to go, respectively. Figure 6.5 summarizes these calculations (Al-Najjar, 1995; Flint et al., 2010).

Figure 6.5(c) shows the optimal solution to the problem. The least possible delay through the network is 18 minutes. To follow the least-cost route, a commuter has to start at the second intersection from the bottom. According to the optimal decisions, or arrows, in the diagram, we see that he should next move down to the bottom-most intersection in column 4. His following decisions should be up, down, up, down, arriving finally at the bottom-most intersection in the last column (Hansen and Zilberstein, 2001; Kraft et al., 2013).

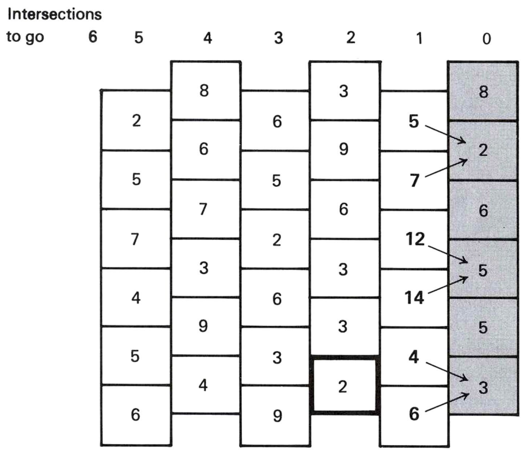


Figure 6.3. Decisions and delays with one intersection to go.

Source: https://www.ime.unicamp.br/~andreani/MS515/capitulo7.pdf.

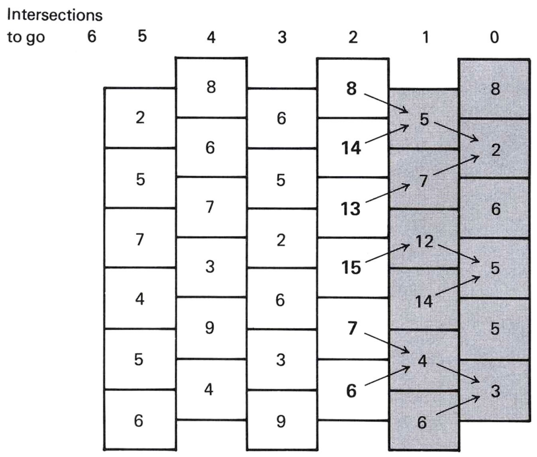


Figure 6.4. Decisions and delays with two intersections to go.

Source: http://web.mit.edu/15.053/www/AMP-Chapter-11.pdf.

- However, the commuters are probably not free to arbitrarily choose the intersection they wish to start from. We can assume that their homes are adjacent to only one of the leftmost *intersections*, and therefore each commuter's starting point is fixed. This assumption does not cause any difficulty since we have, in fact, determined the routes of minimum delay from the downtown parking lots to *all* the commuter's homes (De Moor, 1994; Giegerich, 2000). Note that this assumes that commuters do not care in which downtown lot they park. Instead of solving the minimum-delay problem for only a particular commuter, we have *embedded* the problem of the particular commuter in the more general problem of finding the minimum-delay paths from all homes to the group of downtown parking lots. For example, Figure 6.5 also indicates that the commuter starting at the topmost intersection incurs a delay of 22 minutes if he follows his optimal policy of down, up, up, down, and then down. He presumably parks in a lot close to the second intersection from the top in the last column. Finally, note that three of the intersections in the last column are not entered by any commuter. The analysis has determined the minimum-delay paths from each of the commuter's homes to the group of downtown parking lots, not to each particular parking lot (Karp et al., 1967; Huang, 2008).

Using dynamic programming, we have solved this minimum-delay problem sequentially by keeping track of how many intersections, or stages, there were to go. In dynamic-programming terminology, each point where decisions are made is usually called a *stage* of the decision-making process. At any stage, we need only know which intersection we are in to be able to make subsequent decisions. Our subsequent decisions do not depend upon how we arrived at the particular intersection (Neuneier, 1995; Greene et al., 2020). Information that summarizes the knowledge required about the problem in order to make the current decisions, such as the intersection we are in at a particular stage, is called a *state* of the decision-making process (Giegerich, 2000; Höner et al., 2014).

In terms of these notions, our solution to the minimum-delay problem involved the following intuitive idea, usually referred to as the *principle of optimality*.

Any optimal policy has the property that, whatever the current state and decision, the remaining decisions must constitute an optimal policy with regard to the state resulting from the current decision.

To make this principle more concrete, we can define the *optimal-value function* in the context of the minimum-delay problem.

$v_n(s_n)$ = Optimal value (minimum delay) over the current and subsequent stages (intersections), given that we are in state s_n (in a particular intersection) with n stages (intersections) to go.

The optimal-value function at each stage in the decision-making process is given by the appropriate column of Figure 6.5(c).

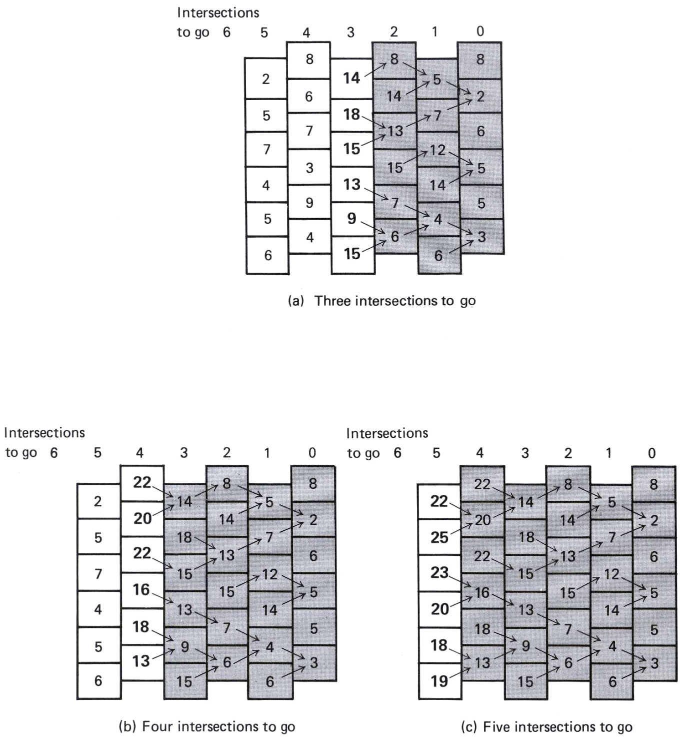

Intersections to go 6 5 4 3 2 1 0
(a) Three intersections to go
(b) Four intersections to go
(c) Five intersections to go

Figure 6.5. Charts of optimal delays and decisions.

Source: https://www.ime.unicamp.br/~andreani/MS515/capitulo7.pdf.

We can write down a *recursive* relationship for computing the optimal-value function by recognizing that, at each stage, the decision in a particular state is determined simply by choosing the minimum total delay (Curtis, 1997; Sauthoff, 2010). If we number the states at each stage as $s_n = 1$ (bottom intersection) up to $s_n = 6$ (top intersection), then:

$$v_n(s_n) = \text{Min}\,\{t_n(s_n) + v_{n-1}(s_{n-1})\},$$

subject to:

$$s_{n-1} = \begin{cases} s_n + 1 & \text{if we choose up and } n \text{ even,} \\ s_n - 1 & \text{if we choose down and } n \text{ odd,} \\ s_n & \text{otherwise,} \end{cases} \quad (1)$$

where; $t_n(s_n)$ is the delay time in intersection s_n at stage n.

The columns of Figure 6.5(c) are then determined by starting at the right while successively applying Eq. (1):

$v_0(s_0) = t_0(s_0)$ $(s_0 = 1, 2, \ldots, 6)$ (2)

Corresponding to this optimal-value function is an *optimal-decision function*, which is simply a list giving the optimal decision for each state at every stage. For this example, the optimal decisions are given by the arrows leaving each box in every column of Figure 6.5(c).

The method of computation illustrated above is called *backward induction*, since it starts at the right and moves back one stage at a time. Its analog, *forward induction*, which is also possible, starts at the left and moves forward one stage at a time (Boutilier et al., 1999; Shin et al., 2019). The spirit of the calculations is identical but the interpretation is somewhat different. The optimal-value function for forward induction is defined by:

$u_n(s_n)$ = Optimal value (minimum delay) over the current and completed stages (intersections), given that we are in state s_n (in a particular intersection) with n stages (intersections) to go.

The recursive relationship for forward induction on the minimum-delay problem is:

$$u_{n-1}(s_{n-1}) = \text{Min}\,\{u_n(s_n) + t_{n-1}(s_{n-1})\}, \quad (3)$$

subject to:

$$s_{n-1} = \begin{cases} s_n + 1 & \text{if we choose up and } n \text{ even,} \\ s_n - 1 & \text{if we choose down and } n \text{ odd,} \\ s_n & \text{otherwise,} \end{cases}$$

where; the stages are numbered in terms of intersections to go. The computations are carried out by setting and successively applying Eqn. (3):

$u_5(s_5) = t_5(s_5)$ $(s_5 = 1, 2, \ldots, 6)$ (4)

The calculations for forward induction are given in Figure 6.6. When performing forward induction, the stages are usually numbered in terms of the number of stages *completed* (rather than the number of stages to go).

However, in order to make a comparison between the two approaches easier, we have avoided using the "stages completed" numbering.

The columns of Figure 6.6(f) give the optimal-value function at each stage for the minimum-delay problem, computed by forward induction. This figure gives the minimum delays from each downtown parking lot to the *group* of homes of the commuters. Therefore, this approach will only guarantee finding the minimum delay path from the downtown parking lots to *one* of the commuters' homes (Rust, 1989; Dorigo et al., 1999). The method, in fact, finds the minimum-delay path to a particular origin only if that origin may be reached from a downtown parking lot by a backward sequence of arrows in Figure 6.6(f).

If we select the minimum-delay path in Figure 6.6(f), lasting 18 minutes, and follow the arrows backward, we discover that this path leads to the intersection second from the bottom in the first column. This is the same minimum-delay path determined by backward induction in Figure 6.5(c).

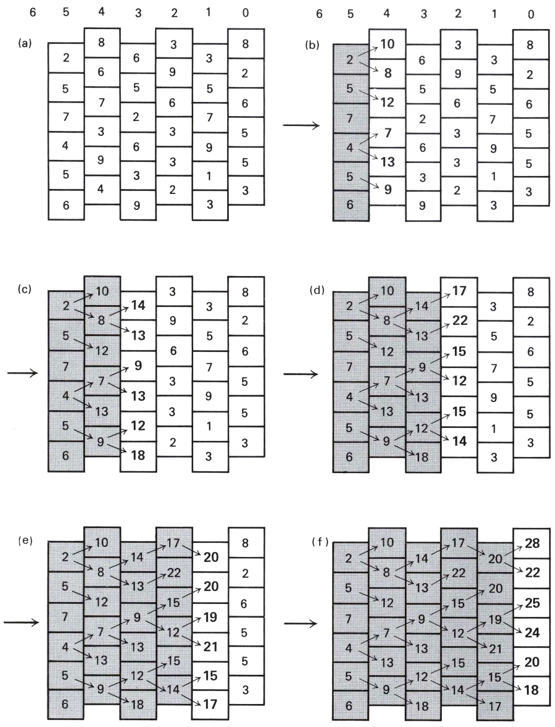


Figure 6.6. Solution by forward induction.

Source: http://web.mit.edu/15.053/www/AMP-Chapter-11.pdf.

Forward induction determined the minimum-delay paths from each individual parking lot to the *group of homes*, while backward induction determined the minimum-delay paths from each individual home to the *group of downtown parking lots*. The minimum-delay path between the two groups is guaranteed to be the same in each case but, in general, the remaining paths determined may be different. Therefore, when using dynamic programming, it is necessary to think about whether forward or backward induction is best suited to the problem you want to solve (Eckstein et al., 1989; Arnold et al., 1993).

6.3. FORMALIZING THE DYNAMIC-PROGRAMMING APPROACH

The elementary example presented in the previous section illustrates the three most important characteristics of dynamic-programming problems.

6.3.1. Stages

The essential feature of the dynamic-programming approach is the structuring of optimization problems into multiple *stages*, which are solved sequentially one stage at a time. Although each one-stage problem is solved as an ordinary optimization problem, its solution helps to define the characteristics of the next one-stage problem in the sequence (Bush et al., 2000; Westphal et al., 2003).

Often, the stages represent different time periods in the problem's planning horizon. For example, the problem of determining the level of inventory of a single commodity can be stated as a dynamic program. The decision variable is the amount to order at the beginning of each month; the objective is to minimize the total ordering and inventory-carrying costs; the basic constraint requires that the demand for the product be satisfied. If we can order only at the beginning of each month and we want an optimal ordering policy for the coming year, we could decompose the problem into 12 stages, each representing the ordering decision at the beginning of the corresponding month (Boutilier et al., 2000; Lewis et al., 2013).

Sometimes the stages do not have time implications. For example, in the simple situation presented in the preceding section, the problem of determining the routes of minimum delay from the homes of the commuters to the downtown parking lots was formulated as a dynamic program. The decision variable was whether to choose *up* or *down* in any intersection, and the stages of the process were defined to be the number of intersections to go. Problems that can be formulated as dynamic programs with stages that do not have time implications are often difficult to recognize (Mitten, 1974; Powell, 2010).

6.3.2. States

Associated with each stage of the optimization problem are the *states* of the process. The states reflect the information required to fully assess the consequences that the current decision has upon future actions. In the inventory problem given in this section, each stage has only one variable describing the state: the inventory level on hand of the single commodity (Barnett et al., 2004; Johannesson et al., 2007). The minimum-delay problem also has one state variable: the intersection a commuter is in at a particular stage.

The specification of the states of the system is perhaps the most critical design parameter of the dynamic programming model (Hunt, 1963; Sutton et al., 1992). There are no set rules for doing this. In fact, for the most part, this is an art often requiring creativity and subtle insight about the problem being studied. The essential properties that should motivate the selection of states are:

- The states should convey enough information to make future decisions without regard to how the process reached the current state; and
- The number of state variables should be small, since the computational effort associated with the dynamic-programming approach is prohibitively expensive when there are more than two, or possibly three, state variables involved in the model formulation.

This last feature considerably limits the applicability of dynamic programming in practice.

6.3.3. Recursive Optimization

The final general characteristic of the dynamic-programming approach is the development of a *recursive optimization* procedure, which builds to a solution of the overall N-stage problem by first solving a one-stage problem and sequentially including one stage at a time and solving one-stage problems until the overall optimum has been found. This procedure can be based on a *backward induction* process, where the first stage to be analyzed is the final stage of the problem and problems are solved moving back one stage at a time until all stages are included. Alternatively, the recursive procedure can be based on a *forward induction* process, where the first stage to be solved is the initial stage of the problem and problems are solved moving forward one stage at a time, until all stages are included (Cohen, 1981; Goguen et al., 1992). In certain problem settings, only one of these induction processes can be applied (e.g., only backward induction is allowed in most problems involving uncertainties).

The basis of the recursive optimization procedure is the so-called *principle of optimality*, which has already been stated: an optimal policy has the property that, whatever the current state and decision, the remaining decisions must constitute an optimal policy with regard to the state resulting from the current decision (Nolan et al., 1972; Gratton et al., 2008).

6.3.4. General Discussion

In what follows, we will formalize the ideas presented thus far. Suppose we have a multistage decision process where the *return* (or cost) for a particular *stage* is:

$$f_n(d_n, s_n) \qquad (5)$$

where; d_n is a permissible *decision* that may be chosen from the set D_n; and s_n is the *state* of the process with n stages to go. Normally, the set of feasible decisions, D_n, available at a given stage depends upon the state of the process at that stage, s_n, and could be written formally as $D_n(s_n)$. To simplify our presentation, we will denote the set of feasible decisions simply as D_n. Now, suppose that there is a total of N stages in the process and we continue to think of n as the number of stages *remaining* in the process. Necessarily, this view implies a finite number of stages in the decision process and therefore a specific horizon for a problem involving time. Further, we assume that the state s_n of the system with n stages to go is a full description of the system for decision-making purposes and that knowledge of prior states is unnecessary. The next state of the process depends entirely on the current state of the process and the current decision taken (Gelfand et al., 1991; Pil et al., 1996). That is, we can define a *transition function* such that, given s_n, the state of the process with n stages to go, the subsequent state of the process with $(n - 1)$ stages to go is given by:

$$s_{n-1} = t_n(d_n, s_n) \qquad (6)$$

where; d_n is the decision chosen for the current stage and state. Note that there is no uncertainty as to what the next state will be, once the current state and current decision are known. In Section 6.7, we will extend these concepts to include uncertainty in the formulation.

Our multistage decision process can be described by the diagram given in Figure 6.7. Given the current state s_n which is a complete description of the system for decision-making purposes with n stages to go, we want to choose the decision d_n that will maximize the total return over the remaining stages. The decision d_n, which must be chosen from a set D_n of permissible decisions, produces a return at this stage of $f_n(d_n,s_n)$ and results in a new state s_{n-1} with $(n-1)$ stages to go. The new state at the beginning of the next stage is determined by the transition function $s_{n-1} = t_n(d_n,s_n)$, and the new state is a complete description of the system for decision-making purposes with $(n-1)$ stages to go. Note that the stage returns are independent of one another (El Karoui et al., 2001; Ordonez, 2009).

In order to illustrate these rather abstract notions, consider a simple inventory example. In this case, the state s_n of the system is the inventory level I_n with n months to go in the planning horizon. The decision d_n is the amount O_n to order this month. The resulting inventory level I_{n-1} with $(n-1)$ months to go is given by the usual inventory-balance relationship:

$I_{n-1} = I_n + O_n - R_n$

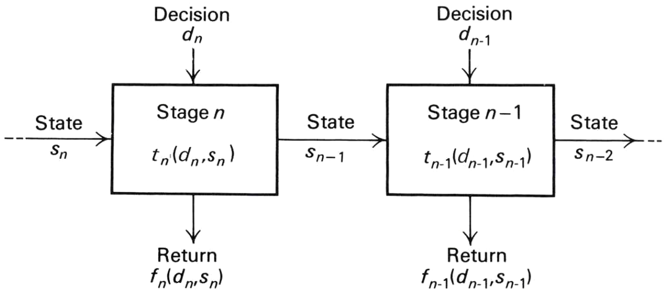


Figure 6.7. Multistage decision process.

Source: https://www.ime.unicamp.br/~andreani/MS515/capitulo7.pdf.

where; R_n is the demand requirement this month. Thus, formally, the transition function with n stages to go is defined to be:

*In–*1 = *tn(In, On)* = *In* + *On* – *Rn.*

The objective to be minimized is the total ordering and inventory-carrying costs, which is the sum of the one-stage costs $C_n(I_n, O_n)$.

6.4. OPTIMAL CAPACITY EXPANSION

In this section, we further illustrate the dynamic-programming approach by solving a problem of optimal capacity expansion in the electric power industry (Kunz and Pradhan, 1994; Sun et al., 2017).

Suppose that a regional electric power company is planning a large investment in nuclear power plants over the next few years. A total of eight nuclear power plants must be built over the next six years because of both increasing demand in the region and the energy crisis, which has forced the closing of certain of their antiquated fossil fuel plants. Suppose that, for a first approximation, we assume that demand for electric power in the region is known with certainty and that we must satisfy the minimum levels of cumulative demand indicated in Table 6.1. The demand here has been converted into equivalent numbers of nuclear power plants required by the end of each year. Due to the extremely adverse public reaction and subsequent difficulties with the public utilities commission, the power company has decided at least to meet this minimum-demand schedule (Basri and Jacobs, 2003; Pavoni et al., 2018).

The building of nuclear power plants takes approximately one year. In addition to a cost directly associated with the construction of a plant, there is a common cost of \$1.5 million incurred when any plants are constructed in any year, independent of the number of plants constructed. This common cost results from contract preparation and certification of the impact statement for the Environmental Protection Agency. In any given year, at most three plants can be constructed. The cost of construction per plant is given in Table 6.1 for each year in the planning horizon. These costs are currently increasing due to the elimination of an investment tax credit designed to speed investment in nuclear power. However, new technology should be available by 1984, which will tend to bring the costs down, even given the elimination of the investment tax credit (Jacobs et al., 2006).

We can structure this problem as a dynamic program by defining the state of the system in terms of the cumulative capacity attained by the end of a particular year. Currently, we have no plants under construction, and by the end of each year in the planning horizon we must have completed a number of plants equal to or greater than the cumulative demand. Further, it is assumed that there is no need ever to construct more than eight plants. Figure 6.8 provides a graph depicting the allowable capacity (states) over time. Any node of this graph is completely described by the corresponding year number and level of cumulative capacity, say the node *(n, p)*. Note that we have chosen to measure time in terms of *years to go* in the planning horizon (Bradford et al., 1971; Wu et al., 2004).

Table 6.1. Demand and Cost per Plant ($ × 1000)

Year	Cumulative Demand (in number of plants)	Cost per Plant ($ × 1000)
1981	1	5,400
1982	2	5,600
1983	4	5,800
1984	6	5,700
1985	7	5,500
1986	8	5,200

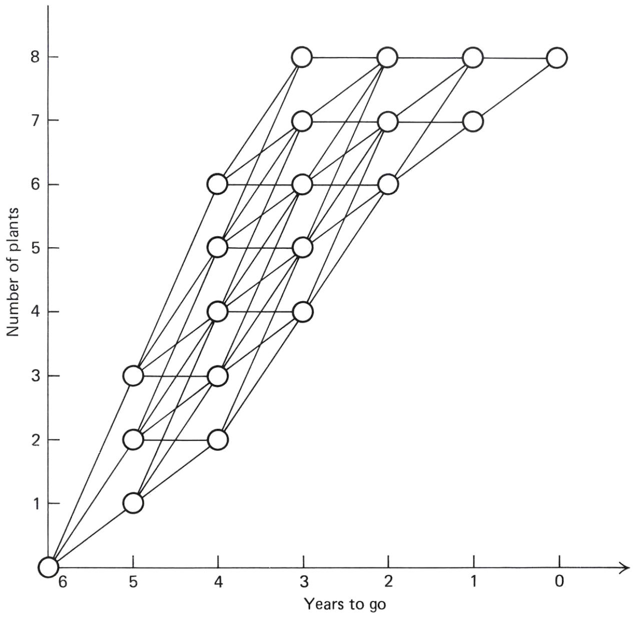


Figure 6.8. Allowable capacity (states) for each stage.

Source: http://web.mit.edu/15.053/www/AMP-Chapter-11.pdf.

The cost of traversing any upward-sloping arc is the common cost of \$1.5 million plus the plant costs, which depend upon the year of construction and whether 1, 2, or 3 plants are completed. Measured in thousands of dollars, these costs are:

$1500 + c_n x_n$

where; c_n is the cost per plant in the year n; and x_n is the number of plants constructed. The cost for traversing any horizontal arc is zero, since these arcs correspond to a situation in which no plant is constructed in the current year (Megiddo, 1984).

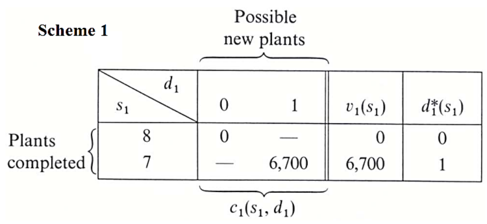
Rather than simply developing the optimal-value function in equation form, as we have done previously, we will perform the identical calculations in Scheme form to highlight the dynamic-programming methodology. To begin, we label the final state zero or, equivalently define the "stage-zero" optimal-value function to be zero for all possible states at stage zero. We will define a state as the cumulative total number of plants completed (Kaplan et al., 1975; Gil et al., 2014). Since the only permissible final state is to construct the entire cumulative demand of eight plants, we have $s_0 = 8$ and,

$v_0(8) = 0$.

Now we can proceed recursively to determine the optimal-value function with one stage remaining. Since the demand data requires 7 plants by 1985, with one year to go the only permissible states are to have completed 7 or 8 plants. We can describe the situation by Scheme 1.

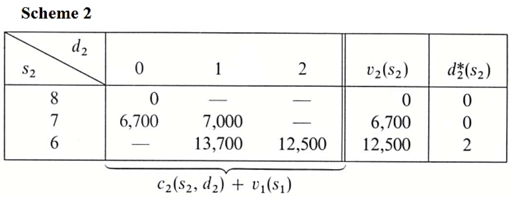
The dashes indicate that the particular combination of current state and decision results in a state that is not permissible. In this table there are no choices, since, if we have not already completed eight plants, we will construct one more to meet the demand. The cost of constructing the one additional plant is the \$1,500 common cost plus the \$5,200 cost per plant, for a total of \$6,700. (All costs are measured in thousands of dollars.) The column headed $d_1^*(s_1)$ gives the optimal decision function, which specifies the optimal number of plants to construct, given the current state of the system (Bickel, 1978; Ahmed et al., 2003).

Now let us consider what action we should take with two years (stages) to go. Scheme 2 indicates the possible costs of each state:



Scheme 1

		Possible new plants			
	s_1 \ d_1	0	1	$v_1(s_1)$	$d_1^*(s_1)$
Plants completed	8	0	—	0	0
	7	—	6,700	6,700	1
		$c_1(s_1, d_1)$			

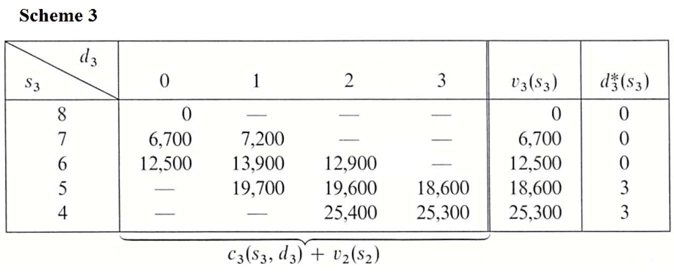


Scheme 2

s_2 \ d_2	0	1	2	$v_2(s_2)$	$d_2^*(s_2)$
8	0	—	—	0	0
7	6,700	7,000	—	6,700	0
6	—	13,700	12,500	12,500	2
	$c_2(s_2, d_2) + v_1(s_1)$				

If we have already completed eight plants with two years to go, then clearly, we will not construct any more. If we have already completed seven plants with two years to go, then we can either construct the one plant we need this year or postpone its construction. Constructing the plant now costs \$1,500 in common costs plus \$5,500 in variable costs, and results in state 8 with one year to go (Sherali et al., 1982; Schapire et al., 1999). Since the cost of state 8 with one year to go is zero, the total cost over the last two years is \$7,000. On the other hand, delaying construction costs zero this year and results in state 7 with one year to go. Since the cost of state 7 with one year to go is \$6,700, the total cost over the last two years is \$6,700. If we arrive at the point where we have two years to go and have completed seven plants, it pays to delay the production of the last plant needed. In a similar way, we can determine that the optimal decision when in state 6 with two years to go is to construct two plants during the next year (Myerson, 1982; Guimaraes et al., 2010).

To make sure that these ideas are firmly understood, we will determine the optimal-value function and optimal decision with three years to go. Consider Scheme 3 for three years to go:

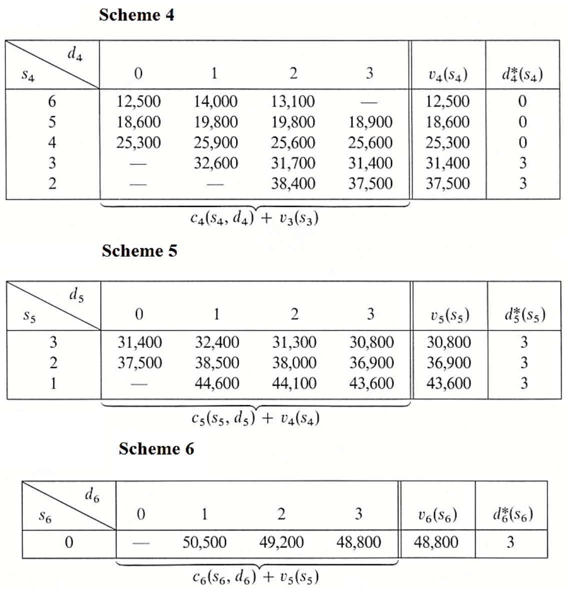


Scheme 3

s_3 \ d_3	0	1	2	3	$v_3(s_3)$	$d_3^*(s_3)$
8	0	—	—	—	0	0
7	6,700	7,200	—	—	6,700	0
6	12,500	13,900	12,900	—	12,500	0
5	—	19,700	19,600	18,600	18,600	3
4	—	—	25,400	25,300	25,300	3
	$c_3(s_3, d_3) + v_2(s_2)$					

Now suppose that, with three years to go, we have completed five plants. We need to construct at least one plant this year in order to meet demand. In fact, we can construct either 1, 2, or 3 plants. If we construct one plant, it costs \$1,500 in common costs plus \$5,700 in plant costs, and results in state 6 with two years to go (Shier, 1979). Since the minimum cost following the optimal policy for the remaining two years is then \$12,500, our total cost for three years would be \$19,700. If we construct two plants, it costs the \$1,500 in common costs plus \$11,400 in plant costs and results in state 7 with two years to go. Since the minimum cost following the optimal policy for the remaining two years is then \$6,700, our total cost for three years would be \$19,600. Finally, if we construct three plants, it costs the \$1,500 in common costs plus \$17,100 in plant costs and results in state 8 with two years to go (Shier, 1976; Sung et al., 2000). Since the minimum cost following the optimal policy for the remaining two years is then zero, our total cost for three years would be \$18,600.

Hence, the optimal decision, having completed five plants (being in state 5) with three years (stages) to go, is to construct three plants this year. The remaining Schemes for the entire dynamic-programming solution are determined in a similar manner (see Figure 6.9).



Scheme 4

s_4 \ d_4	0	1	2	3	$v_4(s_4)$	$d_4^*(s_4)$
6	12,500	14,000	13,100	—	12,500	0
5	18,600	19,800	19,800	18,900	18,600	0
4	25,300	25,900	25,600	25,600	25,300	0
3	—	32,600	31,700	31,400	31,400	3
2	—	—	38,400	37,500	37,500	3

$c_4(s_4, d_4) + v_3(s_3)$

Scheme 5

s_5 \ d_5	0	1	2	3	$v_5(s_5)$	$d_5^*(s_5)$
3	31,400	32,400	31,300	30,800	30,800	3
2	37,500	38,500	38,000	36,900	36,900	3
1	—	44,600	44,100	43,600	43,600	3

$c_5(s_5, d_5) + v_4(s_4)$

Scheme 6

s_6 \ d_6	0	1	2	3	$v_6(s_6)$	$d_6^*(s_6)$
0	—	50,500	49,200	48,800	48,800	3

$c_6(s_6, d_6) + v_5(s_5)$

Figure 6.9. Tables to complete power-plant example.

Source: https://www.ime.unicamp.br/~andreani/MS515/capitulo7.pdf.

Since we start the construction process with no plants (i.e., in state 0) with six years (stages) to go, we can proceed to determine the optimal sequence of decisions by considering the Schemes in the reverse order (Azevedo et al., 1993; Hamed, 2010). With six years to go it is optimal to construct three plants, resulting in state 3 with five years to go. It is then optimal to construct three plants, resulting in state 6 with four years to go, and so forth. The optimal policy is then shown in the tabulation below:

Years to Go	**Construct**	**Resulting State**
6	3	3
5	3	6
4	0	6
3	0	6
2	2	8
1	0	8

Hence, from Scheme 6, the total cost of the policy is \$48.8 million.

6.5. DISCOUNTING FUTURE RETURNS

In the example on optimal capacity expansion presented in the previous section, a very legitimate objection might be raised that the *present value of money* should have been taken into account in finding the optimal construction schedule. The issue here is simply that a dollar received today is clearly worth more than a dollar received one year from now, since the dollar received today could be invested to yield some additional return over the intervening year. It turns out that dynamic programming is extremely well suited to take this into account (Hu, 1968; Chan et al., 2001).

We will define, in the usual way, the one-period *discount factor β* as the present value of one dollar received *one period from now*. In terms of interest rates, if the interest rate for the period were i, then one dollar invested now would accumulate to $(1 + i)$ at the end of one period. To see the relationship between the discount factor β and the interest rate i, we ask the question "How much must be invested now to yield one dollar one period from now?" This amount is clearly the present value of a dollar received one period from now, so that $\beta(1 + i) = 1$ determines the relationship between β and i, namely, $\beta = 1/(1 + i)$. If we invest one dollar now for n periods at an interest rate per period of i, then the accumulated value at the end of n periods, assuming the interest is compounded, is $(1 + i)^n$. Therefore, the present value of one dollar received *n* periods from now is $1/(1 + i)^n$ or, equivalently, β^n (Murdoch et al., 1998; Rezaee et al., 2012).

The concept of discounting can be incorporated into the dynamic-programming framework very easily since we often have a return per period (stage) that we may wish to discount by the per-period discount factor (Tierney, 1996; Tao et al., 2006).

6.6. SHORTEST PATHS IN A NETWORK

Although we have not emphasized this fact, dynamic-programming, and shortest-path problems are very similar. In fact, as illustrated by Figures 6.1 and 6.8, our previous examples of dynamic programming can both be interpreted as shortest-path problems.

In Figure 6.8, we wish to move through the network from the starting node (initial state) at stage 6, with no plants yet constructed, to the end node (final state) at stage 0 with eight plants constructed. Every path in the network specifies a strategy indicating how many new plants to construct each year (Johansson et al., 1999; Sharma et al., 2019).

Since the cost of a strategy sums the cost at each stage, the total cost corresponds to the "length" of a path from the starting to end nodes. The minimum-cost strategy then is just the shortest path.

Figure 6.10 illustrates a shortest-path network for the minimum-delay problem. The numbers next to the arcs are delay times. An end node representing the group of downtown parking lots has been added. This emphasizes the fact that we have assumed that the commuters do not care in which lot they park. A start node has also been added to illustrate that the dynamic-programming solution by *backward* induction finds the shortest path from the end node to the start node. In fact, it finds the shortest paths from the end node to *all* nodes in the network, thereby solving the minimum-delay problem for each commuter. On the other hand, the dynamic-programming solution by *forward* induction finds the shortest path from the start node to the end node. Although the *shortest path* will be the same for both methods, forward induction will *not* solve the minimum-delay problem for *all* commuters, since the commuters are not indifferent to which home they arrive (Wang et al., 2002; Dexter et al., 2021).

To complete the equivalence that we have suggested between dynamic programming and shortest paths, we next show how shortest-path problems can be solved by dynamic programming. Actually, several different dynamic-programming solutions can be given, depending upon the structure of the network under study. As a general rule, the more *structured* the network, the more efficient the algorithm that can be developed (Simon, 1956; Cristobal et al., 2009) To illustrate this point we give two separate algorithms applicable to the following types of networks:

- **Acyclic Networks:** These networks contain no directed cycles. That is, we cannot start from any node and follow the arcs in their given directions to return to the same node.
- **Networks without Negative Cycles:** These networks may contain cycles, but the distance around any cycle (i.e., the sum of the lengths of its arcs) must be nonnegative.

In the first case, to take advantage of the acyclic structure of the network, we order the nodes so that, if the network contains the arc *i–j*, then $i > j$. To obtain such an ordering, begin with the terminal node, which can be thought of as having only entering arcs, and number it "one." Then ignore that node and the incident arcs, and number any node that has only incoming arcs as the next node. Since the network is acyclic, there must be such a node. (Otherwise, from any node, we can move along an arc to another node (Huang et al., 1994; Rust, 1996). Starting from any node and continuing to move away from any node encountered, we eventually would revisit a node, determining a cycle, contradicting the acyclic assumption).

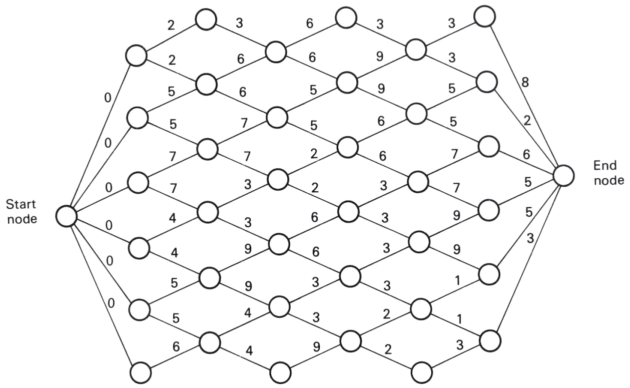


Figure 6.10. Shortest-path network for minimum-delay problem.

Source: https://www.ime.unicamp.br/~andreani/MS515/capitulo7.pdf.

By ignoring the numbered nodes and their incident arcs, the procedure is continued until all nodes are numbered (Sahinidis, 2004; Silva et al., 2020).

We can apply the dynamic-programming approach by viewing each node as a stage, using either backward induction to consider the nodes in ascending order, or forward induction to consider the nodes in reverse order (Bertsekas et al., 1995; Berg et al., 2017). For backward induction, v_n will be interpreted as the longest distance from node n to the end node. Setting $v_1 = 0$, dynamic programming determines $v_2, v_3, \ldots, v_N$ in order, by the recursion

$$v_n = \text{Max}[d_{nj} + v_j] \qquad j < n$$

where; d_{nj} is the given distance on arc n–j. The results of this procedure are given as node labels in Figure 6.11 for the critical-path example.

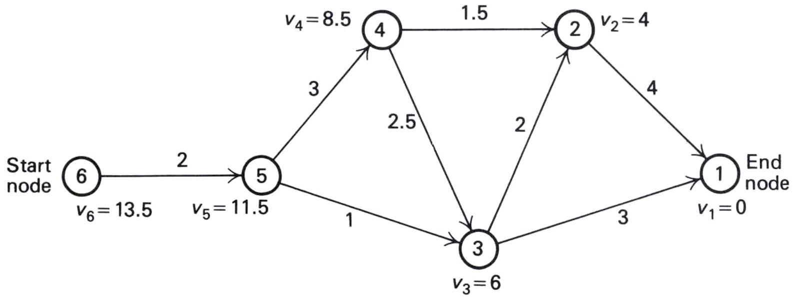


Figure 6.11. Finding the longest path in an acyclic network.

Source: http://web.mit.edu/15.053/www/AMP-Chapter-11.pdf.

For a shortest-path problem, we use minimization instead of maximization. Note that the algorithm finds the longest (shortest) paths from every node to the end node. If we want only the longest path to the start node, we can terminate the procedure once the start node has been labeled. Finally, we could have found the longest distances from the start node to all other nodes by labeling the nodes in the reverse order, beginning with the start node (Figure 6.12) (Sahinidis, 2004; Topaloglu et al., 2006).

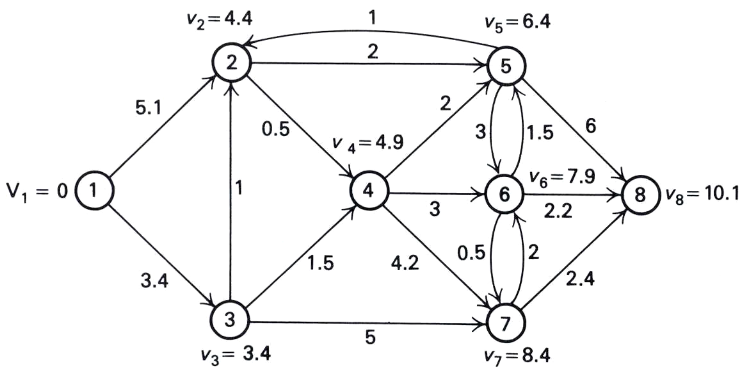


Figure 6.12. Shortest paths in a network without negative cycles.

Source: https://www.ime.unicamp.br/~andreani/MS515/capitulo7.pdf.

A more complicated algorithm must be given for the more general problem of finding the shortest path between two nodes, say nodes 1 and N, in a network without negative cycles. In this case, we can devise a dynamic-programming algorithm based upon a value function defined as follows:

$v_n(j)$ = Shortest distance from node 1 to node j along paths using at most n intermediate nodes.

By definition, then:

$v_0(j) = d_{1j} \quad \text{for } j = 2, 3, \ldots, N$

the length d_{1j} of arc 1–j since no intermediate nodes are used. The dynamic-programming recursion is:

$$v_n(j) = \text{Min}\ \{d_{ij} + v_{n-1}(i)\} \qquad 1 \leq j \leq N \tag{7}$$

which uses the principle of optimality: that any path from node 1 to node j, using at most n intermediate nodes, arrives at node j from node i along arc i–j after using the shortest path with at most $(n - 1)$ intermediate nodes from node j to node i. We allow $i = j$ in the recursion and take $d_{jj} = 0$, since the optimal path using at most n intermediate nodes may coincide with the optimal path with length $v_{n-1}(j)$ using at most $(n - 1)$ intermediate nodes.

The algorithm computes the shortest path from node 1 to every other node in the network. It terminates when $v_n(j) = v_{n-1}(j)$ for every node j, since computations in Eqn. (7) will be repeated at every stage from n on. Because no path (without cycles) uses any more than $(N - 1)$ intermediate nodes, where N is the total number of nodes, the algorithm terminates after at most $(N - 1)$ steps (Xu et al., 2013; Gaggero et al., 2014).

6.7. CONTINUOUS STATE-SPACE PROBLEMS

Until now we have dealt only with problems that have had a finite number of states associated with each stage. Since we also have assumed a finite number of stages, these problems have been identical to finding the shortest path through a network with special structure. Since the development of the fundamental recursive relationship of dynamic programming did not depend on having a finite number of states at each stage, here we introduce an example that has a continuous state space and show that the same procedures still apply (Genc et al., 2007; Machemehl et al., 2014).

Suppose that some governmental agency is attempting to perform cost/benefit analysis on its programs in order to determine which programs should receive funding for the next fiscal year. The agency has managed to put together the information in Table 6.2. The benefits of each program have been converted into equivalent tax savings to the public, and the programs have been listed by decreasing benefit-to-cost ratio (Dai et al., 2012; Keles et al., 2022). The agency has taken the position that there will be no partial funding of programs. Either a program *will* be funded at the indicated level or it will *not* be considered for this budget cycle. Suppose that the agency is fairly sure of receiving a budget of $34 million from the state legislature if it makes a good case that the money is being used effectively (Geramifard et al., 2013; Mohammad et al., 2016). Further, suppose that there is some possibility that the budget will be as high as $42 million. How can the agency make the most effective use of its funds at *either* possible budget level?

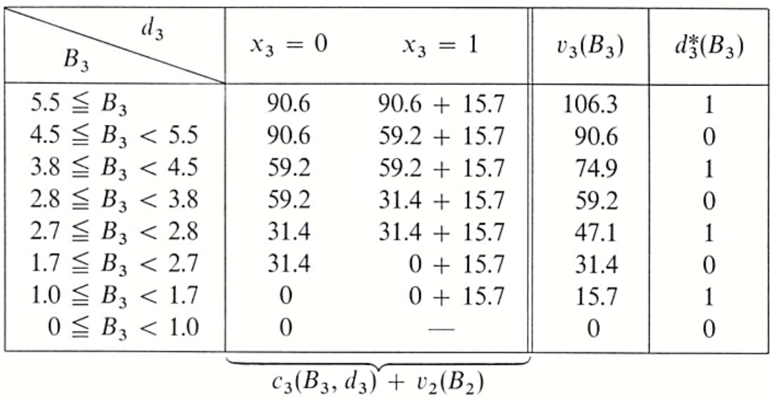
Table 6.2. Cost/Benefit Information by Program

Program	Expected Benefit	Expected Cost	Benefit/Cost
A	$59.2 M	$2.8 M	21.1
B	31.4	1.7	18.4
C	15.7	1.0	15.7
D	30.0	3.2	9.4
E	105.1	15.2	6.9
F	11.6	2.4	4.8
G	67.3	16.0	4.2
H	2.3	0.7	3.3
I	23.2	9.4	2.5
J	18.4	10.1	1.8
	$364.2 M	**$62.5 M**	

We should point out that mathematically this problem is an integer program. If b_j is the benefit of the *j*th program and c_j is the cost of that program, then an integer-programming formulation of the agency's budgeting problem is determined easily (Dantzig, 2004; Webster et al., 2012).

For any budget level, for example, \$4.0 M, we merely consider the two possible decisions: either funding program *C* ($x_3 = 1$) or not ($x_3 = 0$). If we fund program *C*, then we obtain a benefit of \$15.7 M while consuming \$1.0 M of our own budget. The remaining \$3.0 M of our budget is then optimally allocated to the remaining programs, producing a benefit of \$59.2 M, which we obtain from the optimal-value function with the first two programs included. If we do not fund program *C*, then the entire amount of \$4.0 M is optimally allocated to the remaining two programs, producing a benefit of \$59.2. Hence, we should clearly fund program *C* if our budget allocation is \$4.0 M. Optimal decisions taken for other budget levels are determined in a similar manner (Aldasoro et al., 2015; Xie et al., 2017).

Scheme 7



B_3 \ d_3	$x_3 = 0$	$x_3 = 1$	$v_3(B_3)$	$d_3^*(B_3)$
$5.5 \leqq B_3$	90.6	90.6 + 15.7	106.3	1
$4.5 \leqq B_3 < 5.5$	90.6	59.2 + 15.7	90.6	0
$3.8 \leqq B_3 < 4.5$	59.2	59.2 + 15.7	74.9	1
$2.8 \leqq B_3 < 3.8$	59.2	31.4 + 15.7	59.2	0
$2.7 \leqq B_3 < 2.8$	31.4	31.4 + 15.7	47.1	1
$1.7 \leqq B_3 < 2.7$	31.4	0 + 15.7	31.4	0
$1.0 \leqq B_3 < 1.7$	0	0 + 15.7	15.7	1
$0 \leqq B_3 < 1.0$	0	—	0	0

$\underbrace{\qquad\qquad}_{c_3(B_3,\, d_3) + v_2(B_2)}$

Although it is straightforward to continue the recursive calculation of the optimal-value function for succeeding stages, we will not do so since the number of ranges that need to be reported rapidly becomes rather large (Chadès et al., 2014; Liu et al., 2019). The general recursive relationship that determines the optimal-value function at each stage is given by:

$$v_n(B_n) = \text{Max}\ [c_n x_n + v_{n-1}(B_n - c_n x_n)]$$

subject to:

$$x_n = 0 \quad \text{or} \quad 1.$$

The calculation is initialized by observing that:

$$v_0(B_0) = 0$$

for all possible values of B_0. Note that the state transition function is simply:

$Bn - 1 = tn(xn, Bn) = Bn - cnxn.$

We can again illustrate the usual principle of optimality: Given budget B_n at stage n, whatever decision is made with regard to funding the nth program, the remaining budget must be allocated optimally among the first $(n - 1)$ programs. If these calculations were carried to completion, resulting in $v_{10}(B_{10})$ and $d^*_{10}(B_{10})$, then the problem would be solved for all possible budget levels, not just \$3.4 M, and \$4.2 M (Lee et al., 2006; Seuken and Zilberstein, 2007).

Although this example has a continuous state space, a finite number of ranges can be constructed because of the zero–one nature of the decision variables. In fact, all breaks in the range of the state space either are the breaks from the previous stage, or they result from adding the cost of the new program to the breaks in the previous range. This is not a general property of continuous state space problems, and in most cases such ranges cannot be determined. Usually, what is done for continuous state space problems is that they are converted into discrete state problems by defining an appropriate grid on the continuous state space. The optimal-value function is then computed only for the points on the grid. For our cost/benefit example, the total budget must be between zero and \$62.5 M, which provides a range on the state space, although at any stage a tighter upper limit on this range is determined by the sum of the budgets of the first n programs. An appropriate grid would consist of increments of \$0.1 M over the limits of the range at each stage, since this is the accuracy with which the program costs have been estimated. The difference between problems with continuous state spaces and those with discrete state spaces essentially then disappears for computational purposes (Gannon, 1974; Vogstad and Kristoffersen, 2010).

6.8. DYNAMIC PROGRAMMING UNDER UNCERTAINTY

Up to this point we have considered exclusively problems with deterministic behavior. In a deterministic dynamic-programming process, if the system is in state s_n with n stages to go and decision d_n is selected from the set of permissible decisions for this stage and state, then the stage return $f_n(d_n,s_n)$ and the state of the system at the next stage, given by $s_{n-1} = t_n(d_n,s_n)$, are both known with certainty.

This deterministic process can be represented by means of the decision tree in Figure 6.13. As one can observe, given the current state, a specific decision leads with complete certainty to a particular state at the next stage. The stage returns are also known with certainty and are associated with the branches of the tree (Bar-Shalom, 1981; Alterovitz et al., 2008).

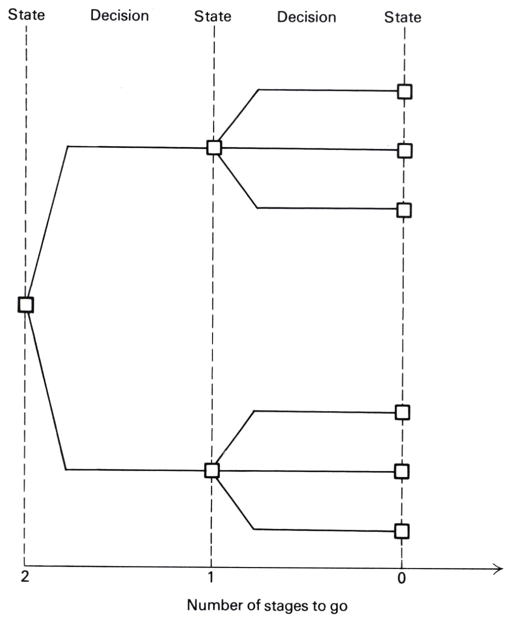


Figure 6.13. Decision tree for deterministic dynamic programming.

Source: http://web.mit.edu/15.053/www/AMP-Chapter-11.pdf.

When uncertainty is present in a dynamic-programming problem, a specific decision for a given state and stage of the process does not, by itself, determine the state of the system at the next stage. Furthermore, this decision may not even determine the return for the current stage. Rather, in dynamic programming under uncertainty, given the state of the system s_n with n stages to go and the current decision d_n, an uncertain event occurs which is determined by a random variable $\tilde{e}_n$ whose outcome e_n is *not* under the control of the decision maker (Zhang et al., 2019; Liu et al., 2020).

The outcomes of the random variable are governed by a probability distribution, $p_n(e_n|d_n,s_n)$, which may be the same for every stage or may be conditional on the stage, the state at the current stage, and even the decision at the current stage.

Figure 6.14 depicts dynamic programming under uncertainty as a *decision tree*, where squares represent states where decisions have to be made and circles represent uncertain events whose outcomes are not under the control of the decision maker. These diagrams can be quite useful in analyzing decisions under uncertainty if the number of possible states is not too large. The decision tree provides a pictorial representation of the sequence of decisions, outcomes, and resulting states, *in the order in which* the decisions must be made and the outcomes become known to the decision maker. Unlike deterministic dynamic programming wherein the optimal decisions at each stage can be specified at the outset, in dynamic programming under uncertainty, the optimal decision at each stage can be selected only after we know the outcome of the uncertain event at the previous stage. At the outset, all that can be specified is a set of decisions that would be made *contingent* on the outcome of a sequence of uncertain events (Huang et al., 2011; Ji et al., 2018).

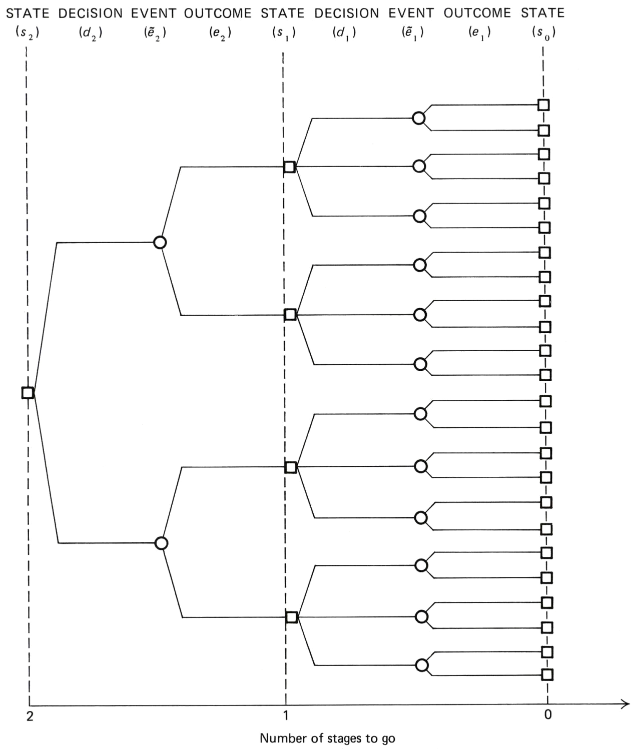


Figure 6.14. Decision tree for dynamic programming under uncertainty.

Source: http://web.mit.edu/15.053/www/AMP-Chapter-11.pdf.

In dynamic programming under uncertainty, since the stage returns and resulting stage may both be uncertain at each stage, we cannot simply optimize the sum of the stage-return functions. Rather, we must optimize the *expected return* over the stages of the problem, taking into account the sequence in which decisions can be made and the outcomes of uncertain events become known to the decision maker. In this situation, backward induction can be applied to determine the optimal strategy, but forward induction cannot. The difficulty with forward induction is that it is impossible to assign values to states at the next stage that are independent of the uncertain evolution of the process from that future state on. With backward induction, on the other hand, no such difficulties arise since the states with zero stages to go are evaluated first, and then the states with one stage to go are evaluated by computing the expected value of any decision and choosing optimally (Barto et al., 1995; Shuai et al., 2018).

We start the backward induction process by computing the optimal-value function at stage 0. This amounts to determining the value of ending in each possible stage with 0 stages to go. This determination may involve an optimization problem or the value of the assets held at the horizon. Next, we compute the optimal-value function at the previous stage. To do this, we first compute the expected value of each uncertain event, weighting the stage return plus the value of the resulting state for each outcome by the probability of each outcome. Then, for each state at the previous stage, we select the decision that has the maximum (or minimum) expected value. Once the optimal-value function for stage 1 has been determined, we continue in a similar manner to determine the optimal-value functions at prior stages by backward induction (Powell et al., 2005; Firdausiyah et al., 2019).

We can make these ideas more concrete by considering a simple example. A manager is in charge of the replenishment decisions during the next two months for the inventory of a fairly expensive item. The production cost of the item is \$1,000/unit, and its selling price is \$2,000/unit. There is an inventory-carrying cost of \$100/unit per month on each unit left over at the end of the month. We assume there is no setup cost associated with running a production order, and further that the production process has a short lead time; therefore, any amount produced during a given month is available to satisfy the demand during that month. At the present time, there is no inventory on hand. Any inventory left at the end of the next two months has to be disposed of at a salvage value of \$500/unit.

The demand for the item is uncertain, but its probability distribution is identical for each of the coming two months. The probability distribution of the demand is as follows:

Demand	**Probability**
0	0.25
1	0.40
2	0.20
3	0.15

The issue to be resolved is how many units to produce during the first month and, *depending on the actual demand in the first month*, how many units to produce during the second month. Since demand is uncertain, the inventory at the end of each month is also uncertain. In fact, demand could exceed the available units on hand in any month, in which case all excess demand results in lost sales. Consequently, our production decision must find the proper balance between production costs, lost sales, and final inventory salvage value (Costa and Kariniotakis, 2007).

The states for this type of problem are usually represented by the inventory level I_n at the beginning of each month. Moreover, the problem is characterized as a two-stage problem, since there are two months involved in the inventory-replenishment decision. To determine the optimal-value function, let:

$v_n(I_n)$ = Maximum contribution, given that we have I_n units of inventory with n stages to go.

We initiate the backward induction procedure by determining the optimal-value function with 0 stages to go. Since the salvage value is \$500/ unit, we have:

I_0	$v_0(I_0)$
0	0
1	500
2	1,000
3	1,500

To compute the optimal-value function with one stage to go, we need to determine, for each inventory level (state), the corresponding contribution associated with each possible production amount (decision) and level of sales (outcome). For each inventory level, we select the production amount that maximizes the expected contribution.

Table 6.3 provides all the necessary detailed computations to determine the optimal-value function with one stage to go. Column 1 gives the state (inventory level) of the process with one stage to go. Column 2 gives the possible decisions (amount to produce) for each state, and, since demand cannot be greater than three, the amount produced is at most three. Column 3 gives the possible outcomes for the uncertain level of sales for each decision and current state, and column 4 gives the probability of each of these possible outcomes. Note that, in any period, it is impossible to sell more than the supply, which is the sum of the inventory currently on hand plus the amount produced. Hence, the probability distribution of sales differs from that of demand since, whenever demand exceeds supply, the entire supply is sold and the excess demand is lost. Column 5 is the resulting state, given that we currently have I_1 on hand, produce d_1, and sell s_1. The transition function in general is just:

$\tilde{\Gamma}_{n-1} = In + dn - \tilde{s}n$

where; the tildes *(~)* indicate that the level of sales is uncertain and, hence, the resulting state is also uncertain. Columns 6, 7, and 8 reflect the revenue and costs for each state, decision, and sales level, and column 9 reflects the value of being in the resulting state at the next stage. Column 10 merely weights the sum of columns 6 through 9 by the probability of their occurring, which is an intermediate calculation in determining the expected value of making a particular decision, given the current state. Column 11 is then just this expected value; and the asterisk indicates the optimal decision for each possible state.

Table 6.3. Computation of Optimal-Value Function with One Stage to Go

(1)	**(2)**	**(3)**	**(4)**	**(5)**	**(6)**	**(7)**	**(8)**	**(9)**	**(10)**	**(11)**
State I_1	**Produce d_1**	**Sell s_1**	**Probability (=**	**Resulting State**	**Production Cost**	**Sales Revenue**	**Inventory Cost**	**$V_0(I_0)$**	**Probability × \$**	**Expected Contribution**
0	0	0	1.	0	0	0	0	0	0	0
	1	0 1	0.25 0.75	1 0	–1,000 –1,000	0 2,000	–100 0	500 0	–150 750	600
	2	0 1 2	0.25 0.40 0.35	2 1 0	–2,000 –2,000 –2,000	0 2,000 4,000	–200 –100 0	1,000 500 0	–300 160 700	560
	3	0 1 2 3	0.25 0.40 0.20 0.15	3 2 1 0	–3,000 –3,000 –3,000 –3,000	0 2,000 4,000 6,000	–300 –200 –100 0	1,500 1,000 500 0	–450 –80 280 450	200

1	0	0 1	0.25 0.75	1 0	0 0	0 2,000	–100 0	500 0	100 1,500	1600
	1	0 1 2	0.25 0.40 0.35	2 1 0	–1,000 –1,000 –1,000	0 2,000 4,000	–200 –100 0	1,000 500 0	–50 560 1,050	1,560
	2	0 1 2 3	0.25 0.40 0.20 0.15	3 2 1 0	–2,000 –2,000 –2,000 –2,000	0 2,000 4,000 6,000	–300 –200 –100 0	1,500 1,000 500 0	–200 320 480 600	1,200
2	0	0 1 2	0.25 0.40 0.35	2 1 0	0 0 0	0 2,000 4,000	–200 –100 0	1,000 500 0	200 960 1,400	2,560
	1	0 1 2 3	0.25 0.40 0.20 0.15	3 2 1 0	–1,000 –1,000 –1,000 –1,000	0 2,000 4,000 6,000	–300 –200 –100 0	1,500 1,000 500 0	50 720 680 750	2,200
3	0	0 1 2 3	0.25 0.40 0.20 0.15	3 2 1 0	0 0 0 0	0 2,000 4,000 6,000	–300 –200 –100 0	1,500 1,000 500 0	300 1,120 880 900	3,200

The resulting optimal-value function and the corresponding optimal-decision function are determined directly from Table 6.3 and are the following:

I_1	$v_1(I_1)$	$d_1^*(I_1)$
0	600	1
1	1,600	0
2	2,560	0
3	3,200	0

Next, we need to compute the optimal-value function with two stages to go. However, since we have assumed that there is no initial inventory on hand, it is not necessary to describe the optimal-value function for every possible state, but only for $I_2 = 0$. Table 6.4 is similar to Table 6.3 and gives the detailed computations required to evaluate the optimal-value function for this case.

Table 6.4. Computation of Optimal-Value Function with Two Stages to Go, I_2 = 0 Only

(1)	**(2)**	**(3)**	**(4)**	**(5)**	**(6)**	**(7)**	**(8)**	**(9)**	**(10)**	**(11)**
State I_2	**Produce d_2**	**Sell s_2**	**Probability (=**	**Resulting State**	**Production Cost**	**Sales Revenue**	**Inventory Cost**	**$v_1(I_1)$**	**Probability × $**	**Expected Contribu-tion**
0	0	0	1.	0	0	0	0	650	650	650
	1	0 1	0.25 0.75	1 0	–1,000 –1,000	0 2,000	0 0	1,600. 600.	150 1,200	135
	2	0 1 2	0.25 0.40 0.35	2 1 0	–2,000 –2,000 –2,000	0 2,000 4,000	–200 –100 0	2,560 1,600 600	90 600 910	1,600
	3	0 1 2 3	0.25 0.40 0.20 0.15	3 2 1 0	–3,000 –3,000 –3,000 –3,000	0 2,000 4,000 6,000	–300 –200 –100 0	3,200 2,560 1,600. 600.	–25 544 500 540	1,559

The optimal-value function and the corresponding decision function for $I_2 = 0$ are taken directly from Table 6.4 and are the following:

I_2	$v_2(I_2)$	$d_2^*(I_2)$
0	1,600	2

The optimal strategy can be summarized by the decision tree given in Figure 6.14. The expected contribution determined by the dynamic-programming solution corresponds to weighting the contribution of every path in this tree by the *probability* that this path occurs (Kreps and Porteus, 1979). The decision tree in Figure 6.14 emphasizes the contingent nature of the optimal strategy determined by dynamic programming under uncertainty (Deisenroth et al., 2009).

REFERENCES

1. Ahmed, S., King, A. J., & Parija, G., (2003). A multi-stage stochastic integer programming approach for capacity expansion under uncertainty. *Journal of Global Optimization, 26*(1), 3–24.
2. Aldasoro, U., Escudero, L. F., Merino, M., Monge, J. F., & Pérez, G., (2015). On parallelization of a stochastic dynamic programming algorithm for solving large-scale mixed 0–1 problems under uncertainty. *Top, 23*(3), 703–742.
3. Al-Najjar, N., (1995). A theory of forward induction in finitely repeated games. *Theory and Decision, 38*(2), 173–193.
4. Alterovitz, R., Branicky, M., & Goldberg, K., (2008). Motion planning under uncertainty for image-guided medical needle steering. *The International Journal of Robotics Research, 27*(11, 12), 1361–1374.
5. Amini, A. A., Weymouth, T. E., & Jain, R. C., (1990). Using dynamic programming for solving variational problems in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 12*(9), 855–867.
6. Arnold, J., Shaw, S. W., & Pasternack, H. E. N. R. I., (1993). Efficient target tracking using dynamic programming. *IEEE transactions on Aerospace and Electronic Systems, 29*(1), 44–56.
7. Azevedo, J., Costa, M. E. O. S., Madeira, J. J. E. S., & Martins, E. Q. V., (1993). An algorithm for the ranking of shortest paths. *European Journal of Operational Research, 69*(1), 97–106.
8. Barnett, M., Leino, K. R. M., & Schulte, W., (2004). The spec# programming system: An overview. In: *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices* (pp. 49–69). Springer, Berlin, Heidelberg.
9. Bar-Shalom, Y., (1981). Stochastic dynamic programming: Caution and probing. *IEEE Transactions on Automatic Control, 26*(5), 1184–1195.
10. Barto, A. G., Bradtke, S. J., & Singh, S. P., (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence, 72*(1, 2), 81–138.
11. Basri, R., & Jacobs, D. W., (2003). Lambertian reflectance and linear subspaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 25*(2), 218–233.

12. Battigalli, P., & Siniscalchi, M., (2002). Strong belief and forward induction reasoning. *Journal of Economic Theory, 106*(2), 356–391.

13. Bellman, R. E., & Dreyfus, S. E., (2015). *Applied Dynamic Programming* (pp. 10–15). Princeton university press.

14. Berg, J. V. D., Patil, S., & Alterovitz, R., (2017). Motion planning under uncertainty using differential dynamic programming in belief space. In: *Robotics Research* (pp. 473–490). Springer, Cham.

15. Bertsekas, D. P., & Tsitsiklis, J. N., (1995). Neuro-dynamic programming: An overview. In: *Proceedings of 1995 34th IEEE Conference on Decision and Control* (Vol. 1, pp. 560–564). IEEE.

16. Bickel, T. C., (1978). *The Optimal Capacity Expansion of a Chemical Plant Via Nonlinear Integer Programming* (Vol. 2, pp. 100–147). The University of Texas at Austin.

17. Birge, J. R., & Louveaux, F., (2011). *Introduction to Stochastic Programming* (pp. 1–10). Springer Science & Business Media.

18. Boutilier, C., Dean, T., & Hanks, S., (1999). Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research, 11*, 1–94.

19. Boutilier, C., Dearden, R., & Goldszmidt, M., (2000). Stochastic dynamic programming with factored representations. *Artificial Intelligence, 121*(1, 2), 49–107.

20. Bradford, D. F., & Oates, W. E., (1971). Towards a predictive theory of intergovernmental grants. *The American Economic Review, 61*(2), 440–448.

21. Bush, W. R., Pincus, J. D., & Sielaff, D. J., (2000). A static analyzer for finding dynamic programming errors. *Software: Practice and Experience, 30*(7), 775–802.

22. Chadès, I., Chapron, G., Cros, M. J., Garcia, F., & Sabbadin, R., (2014). MDPtoolbox: A multi-platform toolbox to solve stochastic dynamic programming problems. *Ecography, 37*(9), 916–920.

23. Chan, E. P., & Zhang, N., (2001). Finding shortest paths in large network systems. In: *Proceedings of the 9th ACM International Symposium on Advances in Geographic Information Systems* (pp. 160–166).

24. Cohen, J. R., (1981). Segmenting speech using dynamic programming. *The Journal of the Acoustical Society of America, 69*(5), 1430–1438.

25. Costa, L. M., & Kariniotakis, G., (2007). A stochastic dynamic programming model for optimal use of local energy resources in a

market environment. In: *2007 IEEE Lausanne Power Tech* (pp. 449–454). IEEE.

26. Cristobal, M. P., Escudero, L. F., & Monge, J. F., (2009). On stochastic dynamic programming for solving large-scale planning problems under uncertainty. *Computers & Operations Research, 36*(8), 2418–2428.
27. Curtis, S., (1997). Dynamic programming: A different perspective. In: *Algorithmic Languages and Calculi* (pp. 1–23). Springer, Boston, MA.
28. Dai, C., Li, Y. P., & Huang, G. H., (2012). An interval-parameter chance-constrained dynamic programming approach for capacity planning under uncertainty. *Resources, Conservation, and Recycling, 62*, 37–50.
29. Dantzig, G. B., (2004). Linear programming under uncertainty. *Management Science, 50*(12_supplement), 1764–1769.
30. Dayan, P., & Daw, N. D., (2008). Decision theory, reinforcement learning, and the brain. *Cognitive, Affective, & Behavioral Neuroscience, 8*(4), 429–453.
31. De Moor, O., (1994). Categories, relations, and dynamic programming. *Mathematical Structures in Computer Science, 4*(1), 33–69.
32. Deisenroth, M. P., Rasmussen, C. E., & Peters, J., (2009). Gaussian process dynamic programming. *Neurocomputing, 72*(7–9), 1508–1524.
33. Dexter, G., Bello, K., & Honorio, J., (2021). Inverse reinforcement learning in a continuous state space with formal guarantees. *Advances in Neural Information Processing Systems*, 34–40.
34. Dorigo, M., & Di Caro, G., (1999). Ant colony optimization: A new meta-heuristic. In: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)* (Vol. 2, pp. 1470–1477). IEEE.
35. Eckstein, Z., & Wolpin, K. I., (1989). The specification and estimation of dynamic stochastic discrete choice models: A survey. *The Journal of Human Resources, 24*(4), 562–598.
36. El Karoui, N., Peng, S., & Quenez, M. C., (2001). A dynamic maximum principle for the optimization of recursive utilities under constraints. *Annals of Applied Probability,* 664–693.
37. Firdausiyah, N., Taniguchi, E., & Qureshi, A. G., (2019). Modeling city logistics using adaptive dynamic programming based multi-agent simulation. *Transportation Research Part E: Logistics and Transportation Review, 125*, 74–96.

38. Flint, A., Mei, C., Murray, D., & Reid, I., (2010). A dynamic programming approach to reconstructing building interiors. In: *European Conference on Computer Vision* (pp. 394–407). Springer, Berlin, Heidelberg.

39. Gaggero, M., Gnecco, G., & Sanguineti, M., (2014). Approximate dynamic programming for stochastic N-stage optimization with application to optimal consumption under uncertainty. *Computational Optimization and Applications, 58*(1), 31–85.

40. Gannon, C. A., (1974). Optimal intertemporal supply of a public facility under uncertainty: A dynamic programming approach to the problem of planning open space. *Regional and Urban Economics, 4*(1), 25–40.

41. Gelfand, S. B., & Mitter, S. K., (1991). Recursive stochastic algorithms for global optimization in R^d. *SIAM Journal on Control and Optimization, 29*(5), 999–1018.

42. Genc, T. S., Reynolds, S. S., & Sen, S., (2007). Dynamic oligopolistic games under uncertainty: A stochastic programming approach. *Journal of Economic Dynamics and Control, 31*(1), 55–80.

43. Geramifard, A., Walsh, T. J., Tellex, S., Chowdhary, G., Roy, N., & How, J. P., (2013). A tutorial on linear function approximators for dynamic programming and reinforcement learning. *Foundations and Trends® in Machine Learning, 6*(4), 375–451.

44. Giegerich, R., & Meyer, C., (2002). Algebraic dynamic programming. In: *International Conference on Algebraic Methodology and Software Technology* (pp. 349–364). Springer, Berlin, Heidelberg.

45. Giegerich, R., (2000). A systematic approach to dynamic programming in bioinformatics. *Bioinformatics, 16*(8), 665–677.

46. Giegerich, R., (2000). Explaining and controlling ambiguity in dynamic programming. In: *Annual Symposium on Combinatorial Pattern Matching* (pp. 46–59). Springer, Berlin, Heidelberg.

47. Gil, E., Aravena, I., & Cárdenas, R., (2014). Generation capacity expansion planning under hydro uncertainty using stochastic mixed integer programming and scenario reduction. *IEEE Transactions on Power Systems, 30*(4), 1838–1847.

48. Goguen, J. A., & Burstall, R. M., (1992). Institutions: Abstract model theory for specification and programming. *Journal of the ACM (JACM), 39*(1), 95–146.

49. Gratton, S., Sartenaer, A., & Toint, P. L., (2008). Recursive trust-region methods for multiscale nonlinear optimization. *SIAM Journal on Optimization, 19*(1), 414–444.
50. Greene, M. L., Deptula, P., Nivison, S., & Dixon, W. E., (2020). Sparse learning-based approximate dynamic programming with barrier constraints. *IEEE Control Systems Letters, 4*(3), 743–748.
51. Guimaraes, P., & Portugal, P., (2010). A simple feasible procedure to fit models with high-dimensional fixed effects. *The Stata Journal, 10*(4), 628–649.
52. Guo, Y., Ma, J., Xiong, C., Li, X., Zhou, F., & Hao, W., (2019). Joint optimization of vehicle trajectories and intersection controllers with connected automated vehicles: Combined dynamic programming and shooting heuristic approach. *Transportation Research Part C: Emerging Technologies, 98*, 54–72.
53. Hamed, A. Y., (2010). A genetic algorithm for finding the k shortest paths in a network. *Egyptian Informatics Journal, 11*(2), 75–79.
54. Hansen, E. A., & Zilberstein, S., (2001). Monitoring and control of anytime algorithms: A dynamic programming approach. *Artificial Intelligence, 126*(1, 2), 139–157.
55. Hauk, E., & Hurkens, S., (2002). On forward induction and evolutionary and strategic stability. *Journal of Economic Theory, 106*(1), 66–90.
56. Held, M., & Karp, R. M., (1962). A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics, 10*(1), 196–210.
57. Höner Zu, S. C., Prohaska, S. J., & Stadler, P. F., (2014). Dynamic programming for set data types. In: *Brazilian Symposium on Bioinformatics* (pp. 57–64). Springer, Cham.
58. Hu, T. C., (1968). A decomposition algorithm for shortest paths in a network. *Operations Research, 16*(1), 91–102.
59. Huan, X., & Marzouk, Y. M., (2016). *Sequential Bayesian Optimal Experimental Design Via Approximate Dynamic Programming* (pp. 1–25). arXiv preprint arXiv:1604.08320.
60. Huang, G. H., Baetz, B. W., & Patry, G. G., (1994). Grey dynamic programming for waste-management planning under uncertainty. *Journal of Urban Planning and Development, 120*(3), 132–156.
61. Huang, L., (2008). Advanced dynamic programming in semiring and hypergraph frameworks. In: *Coling 2008: Advanced Dynamic*

Programming in Computational Linguistics: Theory, Algorithms, and Applications-Tutorial Notes (pp. 1–18).

62. Huang, T., & Liu, D., (2011). Residential energy system control and management using adaptive dynamic programming. In: *The 2011 International Joint Conference on Neural Networks* (pp. 119–124). IEEE.
63. Hunt, J. A., (1963). *The Optimization of Satellite Reconnaissance by the Application of Dynamic Programming Techniques* (Vol. 1, pp. 16–29). MITRE CORP BEDFORD MA.
64. Jacobs, K., & Steck, D. A., (2006). A straightforward introduction to continuous quantum measurement. *Contemporary Physics, 47*(5), 279–303.
65. Jamal, A., Tauhidur, R. M., Al-Ahmadi, H. M., Ullah, I., & Zahid, M., (2020). Intelligent intersection control for delay optimization: Using meta-heuristic search algorithms. *Sustainability, 12*(5), 1896.
66. Ji, Y., Wang, J., Fang, X., & Zhang, H., (2018). Online optimal operation of microgrid using approximate dynamic programming under uncertain environment. In: *2018 37th Chinese Control Conference (CCC)* (pp. 2235–2241). IEEE.
67. Johannesson, L., Asbogard, M., & Egardt, B., (2007). Assessing the potential of predictive control for hybrid vehicle powertrains using stochastic dynamic programming. *IEEE Transactions on Intelligent Transportation Systems, 8*(1), 71–83.
68. Johansson, R., Verhaegen, M., & Chou, C. T., (1999). Stochastic theory of continuous-time state-space identification. *IEEE Transactions on Signal Processing, 47*(1), 41–51.
69. Kaplan, M. A., & Haimes, Y. Y., (1975). Dynamic programming for optimal capacity expansion of wastewater treatment plants 1. *JAWRA Journal of the American Water Resources Association, 11*(2), 278–293.
70. Kappelman, A. C., & Sinha, A. K., (2021). Optimal control in dynamic food supply chains using big data. *Computers & Operations Research, 126*, 105117.
71. Karp, R. M., & Held, M., (1967). Finite-state processes and dynamic programming. *SIAM Journal on Applied Mathematics, 15*(3), 693–718.
72. Keles, D., & Dehler-Holland, J., (2022). Evaluation of photovoltaic storage systems on energy markets under uncertainty using stochastic dynamic programming. *Energy Economics,* 105800.

73. Kraft, H., & Steffensen, M., (2013). A dynamic programming approach to constrained portfolios. *European Journal of Operational Research, 229*(2), 453–461.

74. Kreps, D. M., & Porteus, E. L., (1979). Dynamic choice theory and dynamic programming. *Econometrica: Journal of the Econometric Society,* 91–100.

75. Kunz, W., & Pradhan, D. K., (1994). Recursive learning: A new implication technique for efficient solutions to CAD problems-test, verification, and optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 13*(9), 1143–1158.

76. Lee, J. H., & Lee, J. M., (2006). Approximate dynamic programming based approach to process control and scheduling. *Computers & Chemical Engineering, 30*(10–12), 1603–1618.

77. Lewis, F. L., & Liu, D., (2013). *Reinforcement Learning and Approximate Dynamic Programming for Feedback Control* (Vol. 17, pp. 5–12). John Wiley & Sons.

78. Li, Z., Elefteriadou, L., & Ranka, S., (2014). Signal control optimization for automated vehicles at isolated signalized intersections. *Transportation Research Part C: Emerging Technologies, 49*, 1–18.

79. Liu, X., Wu, H., Wang, L., & Faqiry, M. N., (2020). Stochastic home energy management system via approximate dynamic programming. *IET Energy Systems Integration, 2*(4), 382–392.

80. Liu, Z., Zhou, Y., Huang, G., & Luo, B., (2019). Risk aversion based inexact stochastic dynamic programming approach for water resources management planning under uncertainty. *Sustainability, 11*(24), 6926.

81. Machemehl, R., Gemar, M., & Brown, L., (2014). A stochastic dynamic programming approach for the equipment replacement optimization under uncertainty. *Journal of Transportation Systems Engineering and Information Technology, 14*(3), 76–84.

82. Megiddo, N., (1984). Linear programming in linear time when the dimension is fixed. *Journal of the ACM (JACM), 31*(1), 114–127.

83. Mitten, L., (1974). Preference order dynamic programming. *Management Science, 21*(1), 43–46.

84. Mohammadghasemi, M., Shahraki, J., & Sabohi, S. M., (2016). *Optimization Model of Hirmand River Basin Water Resources in the Agricultural Sector Using Stochastic Dynamic Programming Under Uncertainty Conditions* (Vol. 3, pp. 150–177).

85. Momoh, J. A., (2009). Smart grid design for efficient and flexible power networks operation and control. In: *2009 IEEE/PES Power Systems Conference and Exposition* (pp. 1–8). IEEE.
86. Murdoch, D. J., & Green, P. J., (1998). Exact sampling from a continuous state space. *Scandinavian Journal of Statistics, 25*(3), 483–502.
87. Myerson, R. B., (1982). Optimal coordination mechanisms in generalized principal-agent problems. *Journal of Mathematical Economics, 10*(1), 67–81.
88. Neuneier, R., (1995). Optimal asset allocation using adaptive dynamic programming. *Advances in Neural Information Processing Systems, 8*, pp. 1–14.
89. Nolan, R. L., & Sovereign, M. G., (1972). A recursive optimization and simulation approach to analysis with an application to transportation systems. *Management Science, 18*(12), B-676.
90. Ordonez, C., (2009). Optimization of linear recursive queries in SQL. *IEEE Transactions on knowledge and Data Engineering, 22*(2), 264–277.
91. Osman, M. S., Abo-Sinna, M. A., & Mousa, A. A., (2005). An effective genetic algorithm approach to multiobjective resource allocation problems (MORAPs). *Applied Mathematics and Computation, 163*(2), 755–768.
92. Pavoni, N., Sleet, C., & Messner, M., (2018). The dual approach to recursive optimization: Theory and examples. *Econometrica, 86*(1), 133–172.
93. Pil, A. C., & Asada, H. H., (1996). Integrated structure/control design of mechatronic systems using a recursive experimental optimization method. *IEEE/ASME Transactions on Mechatronics, 1*(3), 191–203.
94. Powell, W. B., (2010). Approximate dynamic programming-II: Algorithms. *Wiley Encyclopedia of Operations Research and Management Science* (Vol. 2).
95. Powell, W. B., George, A., Bouzaiene-Ayari, B., & Simao, H. P., (2005). Approximate dynamic programming for high dimensional resource allocation problems. In: *Proceedings 2005 IEEE International Joint Conference on Neural Networks* (Vol. 5, pp. 2989–2994). IEEE.
96. Powell, W. B., Shapiro, J. A., & Simão, H. P., (2002). An adaptive dynamic programming algorithm for the heterogeneous resource allocation problem. *Transportation Science, 36*(2), 231–249.

97. Rezaee, K., Abdulhai, B., & Abdelgawad, H., (2012). Application of reinforcement learning with continuous state space to ramp metering in real-world conditions. In: *2012 15th International IEEE Conference on Intelligent Transportation Systems* (pp. 1590–1595). IEEE.

98. Rust, J., (1989). 12 a dynamic programming model of retirement behavior. *The Economics of Aging,* 359.

99. Rust, J., (1996). Numerical dynamic programming in economics. *Handbook of Computational Economics, 1*, 619–729.

100. Rust, J., (2008). Dynamic programming. *The New Palgrave Dictionary of Economics, 1*, 8–15.

101. Sahinidis, N. V., (2004). Optimization under uncertainty: State-of-the-art and opportunities. *Computers & Chemical Engineering, 28*(6, 7), 971–983.

102. Sali, A., & Blundell, T. L., (1990). Definition of general topological equivalence in protein structures: A procedure involving comparison of properties and relationships through simulated annealing and dynamic programming. *Journal of Molecular Biology, 212*(2), 403–428.

103. Sauthoff, G., (2010). *Bellman's GAP: A 2nd Generation Language and System for Algebraic Dynamic Programming*, 15–25).

104. Schapire, R. E., & Singer, Y., (1999). Improved boosting algorithms using confidence-rated predictions. *Machine Learning, 37*(3), 297–336.

105. Sen, S., & Head, K. L., (1997). Controlled optimization of phases at an intersection. *Transportation Science, 31*(1), 5–17.

106. Seuken, S., & Zilberstein, S., (2007). Memory-bounded dynamic programming for DEC-POMDPs. In: *IJCAI* (pp. 2009–2015).

107. Sharma, H., Jain, R., & Gupta, A., (2019). An empirical relative value learning algorithm for non-parametric MDPs with continuous state space. In: *2019 18th European Control Conference (ECC)* (pp. 1368–1373). IEEE.

108. Sherali, H. D., Soyster, A. L., Murphy, F. H., & Sen, S., (1982). Linear programming based analysis of marginal cost pricing in electric utility capacity expansion. *European Journal of Operational Research, 11*(4), 349–360.

109. Shier, D. R., (1976). Iterative methods for determining the k shortest paths in a network. *Networks, 6*(3), 205–229.

110. Shier, D. R., (1979). On algorithms for finding the k shortest paths in a network. *Networks, 9*(3), 195–214.
111. Shin, J., Badgwell, T. A., Liu, K. H., & Lee, J. H., (2019). Reinforcement learning: Overview of recent progress and implications for process control. *Computers & Chemical Engineering, 127*, 282–294.
112. Shuai, H., Fang, J., Ai, X., Tang, Y., Wen, J., & He, H., (2018). Stochastic optimization of economic dispatch for microgrid based on approximate dynamic programming. *IEEE Transactions on Smart Grid, 10*(3), 2440–2452.
113. Silva, T. A., & De Souza, M. C., (2020). Surgical scheduling under uncertainty by approximate dynamic programming. *Omega, 95*, 102066.
114. Simon, H. A., (1956). Dynamic programming under uncertainty with a quadratic criterion function. *Econometrica, Journal of the Econometric Society,* 74–81.
115. Snyder, W. L., Powell, H. D., & Rayburn, J. C., (1987). Dynamic programming approach to unit commitment. *IEEE Transactions on Power Systems, 2*(2), 339–348.
116. Sun, Y., Kirley, M., & Halgamuge, S. K., (2017). A recursive decomposition method for large scale continuous optimization. *IEEE Transactions on Evolutionary Computation, 22*(5), 647–661.
117. Sung, K., Bell, M. G., Seong, M., & Park, S., (2000). Shortest paths in a network with time-dependent flow speeds. *European Journal of Operational Research, 121*(1), 32–39.
118. Sutton, R. S., Barto, A. G., & Williams, R. J., (1992). Reinforcement learning is direct adaptive optimal control. *IEEE Control Systems Magazine, 12*(2), 19–22.
119. Tao, J. Y., & Li, D. S., (2006). Cooperative strategy learning in multi-agent environment with continuous state space. In: *2006 International Conference on Machine Learning and Cybernetics* (pp. 2107–2111). IEEE.
120. Tierney, L., (1996). Introduction to general state-space Markov chain theory. *Markov Chain Monte Carlo in Practice,* 59–74.
121. Topaloglu, H., & Kunnumkal, S., (2006). Approximate dynamic programming methods for an inventory allocation problem under uncertainty. *Naval Research Logistics (NRL), 53*(8), 822–841.

122. Ulmer, M. W., Goodson, J. C., Mattfeld, D. C., & Hennig, M., (2019). Offline–online approximate dynamic programming for dynamic vehicle routing with stochastic requests. *Transportation Science, 53*(1), 185–202.

123. Van, D. E., (1989). Stable equilibria and forward induction. *Journal of Economic Theory, 48*(2), 476–496.

124. Vogstad, K., & Kristoffersen, T. K., (2010). Investment decisions under uncertainty using stochastic dynamic programming: A case study of wind power. In: *Handbook of Power Systems I* (pp. 331–341). Springer, Berlin, Heidelberg.

125. Wang, C. L., & Xie, K. M., (2002). Convergence of a new evolutionary computing algorithm in continuous state space. *International Journal of Computer Mathematics, 79*(1), 27–37.

126. Webster, M., Santen, N., & Parpas, P., (2012). An approximate dynamic programming framework for modeling global climate policy under decision-dependent uncertainty. *Computational Management Science, 9*(3), 339–362.

127. Westphal, M. I., Pickett, M., Getz, W. M., & Possingham, H. P., (2003). The use of stochastic dynamic programming in optimal landscape reconstruction for metapopulations. *Ecological Applications, 13*(2), 543–555.

128. Wu, F., & Huberman, B. A., (2004). Finding communities in linear time: A physics approach. *The European Physical Journal B, 38*(2), 331–338.

129. Xie, J., Wan, Y., & Lewis, F. L., (2017). Strategic air traffic flow management under uncertainties using scalable sampling-based dynamic programming and q-learning approaches. In: *2017 11th Asian Control Conference (ASCC)* (pp. 1116–1121). IEEE.

130. Xu, J., Zeng, Z., Han, B., & Lei, X., (2013). A dynamic programming-based particle swarm optimization algorithm for an inventory management problem under uncertainty. *Engineering Optimization, 45*(7), 851–880.

131. Yagar, S., & Han, B., (1994). A procedure for real-time signal control that considers transit interference and priority. *Transportation Research Part B: Methodological, 28*(4), 315–331.

132. Zhang, N., Leibowicz, B. D., & Hanasusanto, G. A., (2019). Optimal residential battery storage operations using robust data-driven dynamic programming. *IEEE Transactions on Smart Grid, 11*(2), 1771–1780.

CHAPTER 7

FUNDAMENTALS OF OPERATING SYSTEMS

CONTENTS

7.1. INTRODUCTION

An operating system (OS) is a collection of programs that manage the execution of application software and serve as a link between a computer's consumer and its hardware. The OS is software that both maintains computer hardware and offers an environment in which application applications may execute (Eager et al., 2016).

Windows/NT, Windows, MacOS, and OS/2 are instances of OSs.

The operating system goals are as follows (Hughes, 2000):

- To design the computer system user-friendly and simple to operate;
- To make the most use of computer hardware;
- To run user applications and make it simpler to solve user issues.

Application programs, OSs, hardware, and users are the 4 elements that make up a computer system. Figure 7.1 depicts an abstract representation of system elements (Dandamudi, 2003).

- **Users:** These can be thought of as machines, people, or other computers.
- **Hardware:** This includes memory, CPU, and input/output devices.
- **Application Programs:** This includes database systems, compilers, and web browsers, which help users solve their computer challenges.
- **Operating System (OS):** It offers the mechanism for appropriate usage of hardware in computer system operations.

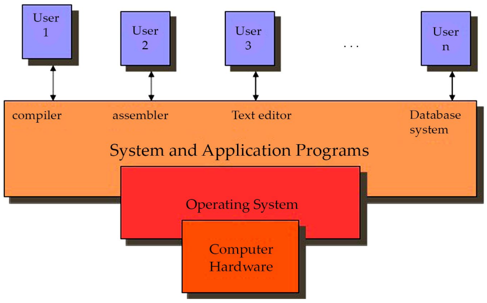


Figure 7.1. Computer system.

Source: https://www.slideshare.net/SHIKHAGAUTAM4/3-basic-organization-of-a-computer.

7.2. COMPUTER SYSTEM ORGANIZATION

7.2.1. Computer-System Operation

A typical computer system is made up of one or many CPUs and multiple device controllers which are all linked via a general bus that allows access to shared memory and other resources (Figure 7.2) (Robey, 1981). It is the responsibility of each device controller to oversee one particular type of equipment (for instance, video displays, audio devices, disc drives). The device controllers and the CPU are capable of running in parallel, with each fighting for memory cycles. A memory controller is in charge of coordinating access to the shared memory to guarantee that it is used in an orderly manner (Estrin, 1960).

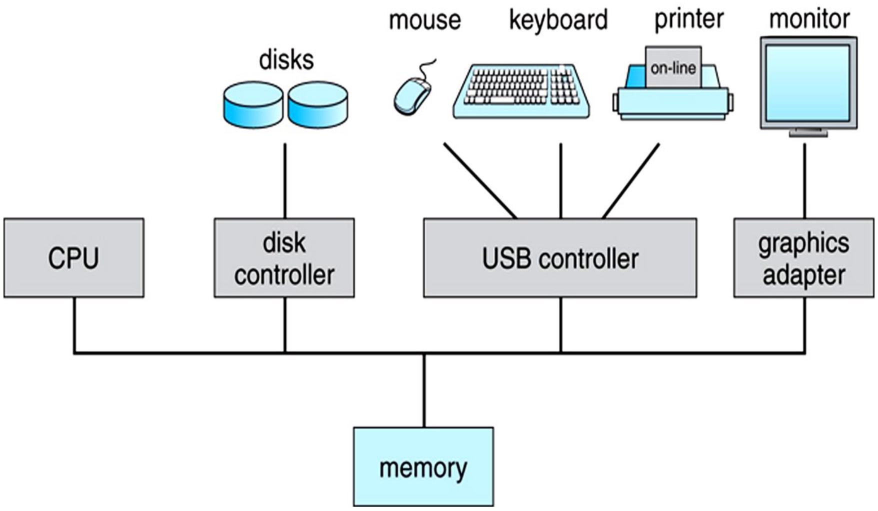


Figure 7.2. A modern computer system.

Source: https://429151971640327878.weebly.com/blog/12-computer-system-organization.

When a computer is first turned on or restarted, it has to execute an initial software to get it up and running. This first software, often known as a bootstrap program, is usually rather simple (Musina et al., 2017). For the most part, firmware is included within the computer hardware, and is saved in read-only memory (ROM) or electronically erasable programmable read-only memory (EEPROM). From the registers of CPU to the device controllers to the contents of the RAM, it configures and configures everything about the system. The bootstrap software should understand how to load the OS and begin running it. To achieve this, the bootstrap software must identify and load the OS kernel into memory. After that, the OS launches the first procedure, like "init," and waits for anything to happen (Austin et al., 2002).

7.2.2. Storage Structure

Computer programs should be stored in primary memory (commonly known as RAM). The only significant storage location that the CPU may access directly is primary memory. It creates a list of memory words. Every word has a unique address. A series of load or store instructions to specified memory locations are used to accomplish interaction. The instruction of load copies a word from primary memory to an interior register in the Central Processing Unit, while the store instruction copies the contents of a register to primary memory (Cardenas, 1973).

The cycle of instruction-execution comprises:

- Retrieves an instruction from memory and places it in the register of instruction. In addition, the register of PC should be incremented;
- Interpret the instruction, which can result in the retrieval of operands from memory and storage in an interior register;
- Run the command and accumulate the output in memory.

For the following 2 causes, the programs and data have not yet remained in the primary memory indefinitely (Murphy et al., 1972):

- The primary memory is a volatile storage medium that eliminates its data when the power is switched off or the device is damaged;
- Main RAM is typically insufficient to hold all required programs. Data is stored indefinitely.

As a result, many computer systems include secondary storage like an extension of primary memory to permanently store vast amounts of data.

A computer system's diverse storage systems may be arranged in a hierarchy (Figure 7.3). Size, speed, volatility, and cost are the key distinctions between the different storage methods. Higher levels are more costly, although they are also quicker (Rosenblum et al., 1992).

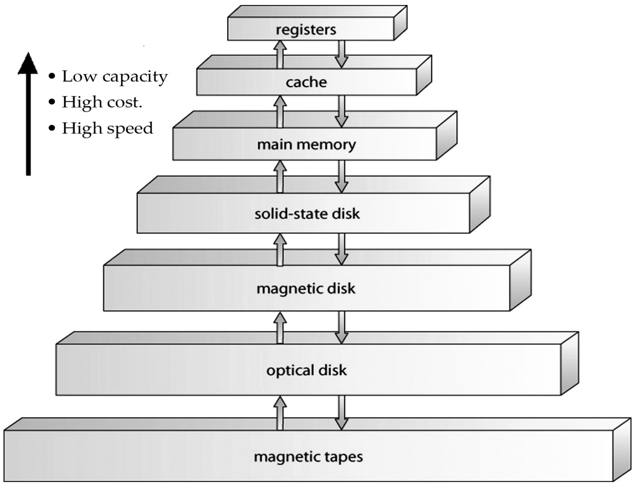


Figure 7.3. Storage device hierarchy.

7.2.3. Input/Output Structure

A computer system is composed of a large number of device controllers and CPUs that are all connected by an ordinary bus (Estrin, 1960). In addition to managing peripheral devices, the device controller has been responsible of transmitting data among these devices and the onboard buffer storage. Many OS give a device driver for each device controller (Figure 7.4).

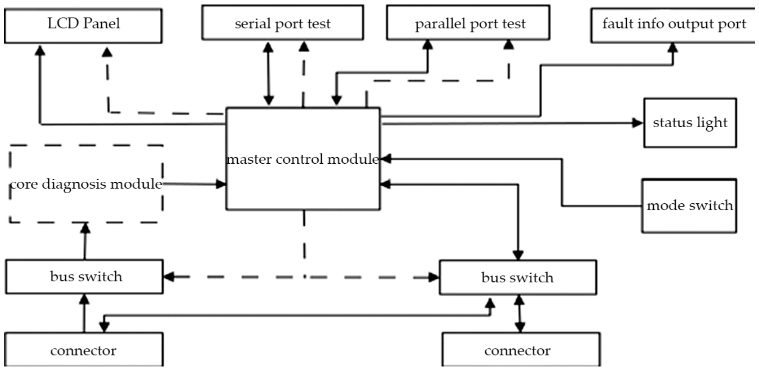


Figure 7.4. Structure of input/output diagnosis module.

To initiate an Input/output operation, the device driver must first load the proper registers inside the device controller. The data of such registers are examined by the device controller to identify which action to take place. The controller initiates the transmit of data from the device to its local buffer by triggering the transfer (Ritchie, 1984). Once a data transfer has been completed, the device controller notifies the device driver through an interrupt that the operation has been completed successfully. Control is subsequently transferred back to the OS via the device driver. Other procedures are completed by returning status information from the device driver.

Direct memory access (DMA) is utilized to transfer large amounts of data. After configuring the Input/output device's buffers, counters, and pointers, the device controller transmits a whole block of data straight to or from its buffer storage to memory, without the need for the CPU to intervene. When using high-speed devices, just one interrupt is created every block to notify the device driver that the operation is finished, as opposed to one interrupt per byte when using lower-speed devices (Haber et al., 1990).

7.3. COMPUTER SYSTEM STRUCTURE

As per the processors quantity utilized, there have been many types for the construction a computer system (Robey, 1981):

- **Single-Processor System:** Only 1 Central Processing Unit is utilized to perform instructions.
  - There are two or many processors share a bus, physical memory, clock, and peripheral devices in a multiprocessor system. Multiprocessors give the benefits as follow:
  - Enhance the throughput;
  - The economic scale (lower cost);
  - Enhance the reliability level.
- **Clustered System:** A clustered system is comprised of a number of computer systems that have been connected together by a local area network.

7.4. OPERATING SYSTEM (OS) HISTORY

Throughout the years, OSs have evolved. The history of OS is seen in the table and Figure 7.5 (Silberschatz et al., 1991).

Generation	Year	Electronic devices used	Types of OS and devices
First	1945 – 55	Vacuum tubes	Plug boards
Second	1955 – 1965	Transistors	Batch system
Third	1965 – 1980	Integrated Circuit (IC)	Multiprogramming
Fourth	Since 1980	Large scale integration	PC

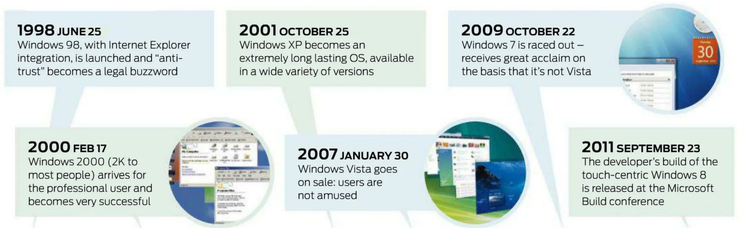

1998 JUNE 25
Windows 98, with Internet Explorer integration, is launched and "anti-trust" becomes a legal buzzword
2001 OCTOBER 25
Windows XP becomes an extremely long lasting OS, available in a wide variety of versions
2009 OCTOBER 22
Windows 7 is raced out – receives great acclaim on the basis that it's not Vista
2000 FEB 17
Windows 2000 (2K to most people) arrives for the professional user and becomes very successful
2007 JANUARY 30
Windows Vista goes on sale: users are not amused
2011 SEPTEMBER 23
The developer's build of the touch-centric Windows 8 is released at the Microsoft Build conference

Figure 7.5. History of operating systems.

Source: https://www.abhishekshukla.com/windows-operating-system/history-evolution-windows-os-operating/.

7.5. OPERATING SYSTEM (OS) FUNCTIONS

The OS executes a variety of tasks, including (Figure 7.6) (Stallings, 2003):

- User interface implementation;
- User-to-user HW sharing;
- Permitting people to share data;
- Avoiding consumers from causing problems for each other;
- Allocating sources to users;
- Streamlining Input/output operations;
- Getting back on track after making a mistake;
- Keeping track of source storage;
- Making parallel processes easier;
- Data organization for safe and quick accessibility;
- Managing network communications is number eleven.

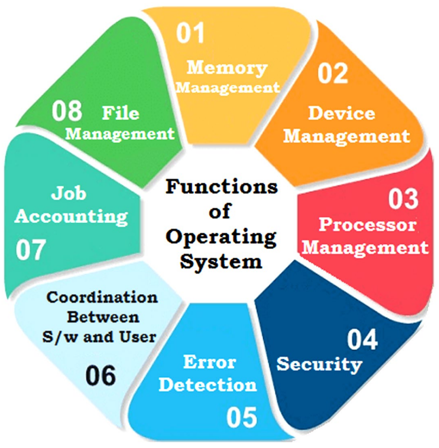


Figure 7.6. Operating system functions.

Source: https://electricalfundablog.com/operating-system-os-functions-types-resource-management/.

7.6. OPERATING SYSTEM (OS) CATEGORIES

The main types of current OSs can be divided into three groups depending upon the type of interaction that occurs among the user and the computer which are discussed in subsections (Figure 7.7) (Agarwal et al., 1988).

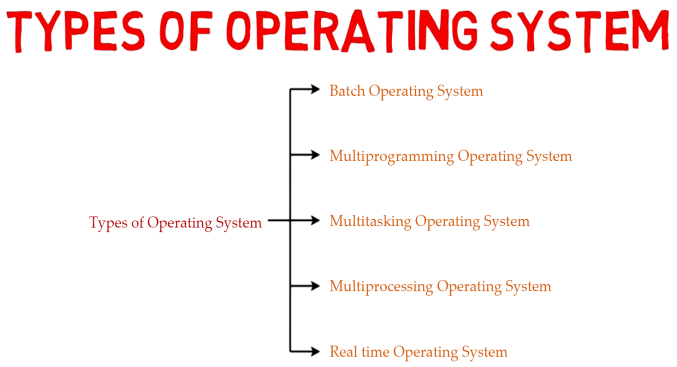

TYPES OF OPERATING SYSTEM
Types of Operating System
Batch Operating System
Multiprogramming Operating System
Multitasking Operating System
Multiprocessing Operating System
Real time Operating System

Figure 7.7. Kinds of operating systems.

Source: https://www.slideserve.com/arleen/operating-systems-for-wireless-sensor-networks-in-space.

7.6.1. Batch System

Users submit work on a normal basis (such as monthly, weekly, and daily) to a central location in this form of OS, and the user of this type of system does not interface directly with the computer system. Jobs with comparable requirements had been grouped and ran through the computer like a group to speed up the processing (Litzkow et al., 1990). As a result, the programmer will hand over control of the programs to the operator. Every job's output will be sent to the relevant programmer. The main duty of this kind was to automatically hand over control from one job to another (Figure 7.8).

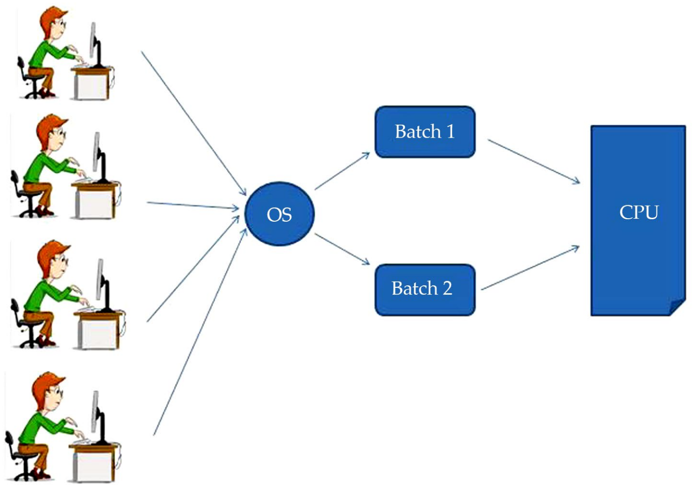


Figure 7.8. A batch operating system is depicted in the diagram.

Source: https://www.techtud.com/short-notes/batch-operating-system.

Batch system drawbacks are given below (Brown et al., 1991):

- From the user's perspective, the turnaround time might be lengthy;
- The program is hard to debug.

7.6.2. Time-Sharing System

This sort of OS allows for an online connection among the consumer and the system, in which the consumer offers direct commands and receives an intermediary response; hence, it is referred to as an interactive system (Ritchie et al., 1978).

The time-sharing technology allows several users to share the computer system at the same time. The Central Processing Unit is swiftly multiplexed between various applications stored in memory and on storage. A program moved back and forth between memory and the disc (Figure 7.9) (Ritchie et al., 1981).

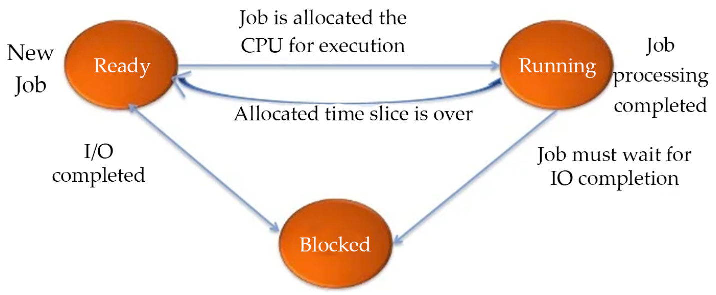

Job is allocated the CPU for execution
New Job
Ready
Running
Job processing completed
Allocated time slice is over
I/O completed
Job must wait for IO completion
Blocked

Figure 7.9. Time-sharing OS's process state diagram.

Source: https://www.slideshare.net/KadianAman/aman-singh.

The CPU optimum time is reduced by using a time-sharing mechanism. The drawback is a little more complicated.

7.6.3. Real-Time OS

A real-time OS is distinguished by its ability to respond quickly. It ensures that time-sensitive jobs are executed on time. For every function to be done on the computer, this kind should have a defined maximum time restriction. Real-time systems have been utilized when there have been tight time limits on the operation of a processor or the data flow, and real-time systems may also be used as a control device in a specific application when strict time constraints are required (Figure 7.10) (Clark et al., 1992).

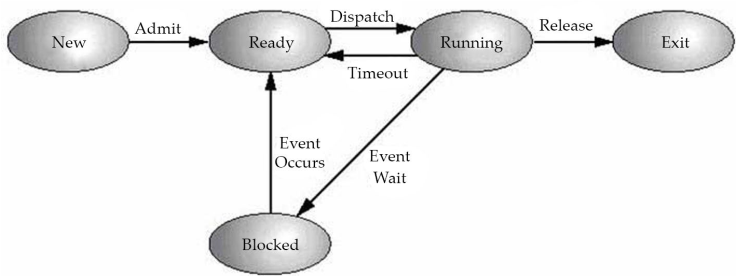


Figure 7.10. The schematic diagram for the real-time operating system.

Source: https://www.polytechnichub.com/rtos-real-time-operating-system/.

This sort of system is exemplified by the airline reservation system.

7.7. THE PERFORMANCE DEVELOPMENT OF OS

7.7.1. Online and Offline Operation

For every Input/output device, a separate function called a device controller was built. Certain Input/output devices are designed to work either online (when linked to the CPU) or off-line (when not connected to the processor) (A control unit is in charge of them) (Figures 7.11 and 7.12) (Seltzer et al., 1997).

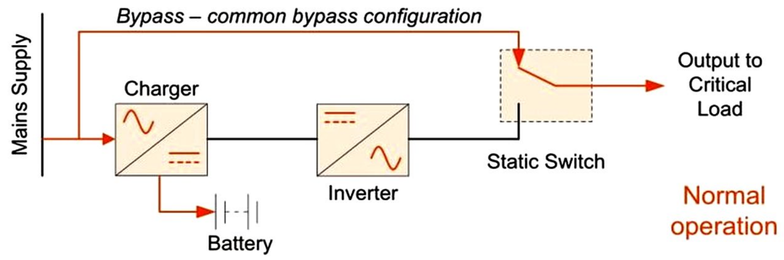


Figure 7.11. Off-line UPS topology.

Source: https://www.datacenterdynamics.com/en/opinions/ups-terminology-101-online-and-offline-ups-topologies/.

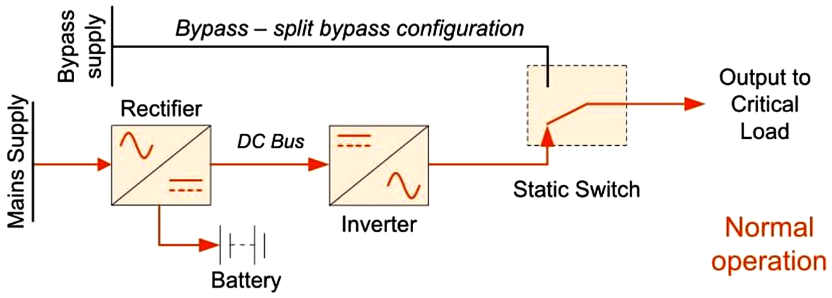


Figure 7.12. Online UPS topology.

Source: https://www.datacenterdynamics.com/en/opinions/ups-terminology-101-online-and-offline-ups-topologies/.

7.7.2. Buffering

A buffer is a major storage space used to retain data during input/output transfers (Hildebrand, 1992). The data are deposited in the buffer via an Input/output channel on input, and the data can be accessible by the CPU once the transfer is complete. Single or double buffing is possible.

7.7.3. Spooling (Simultaneously Peripheral Operation Online)

Spooling makes advantage of the disc as a massive buffer. Since devices access data at varying speeds, spooling is essential. The buffer serves as a holding area for data until the slower device catches up (Lange et al., 2010). Spooling permits you to overlap the calculation of one task with the input/output of another.

7.7.4. Multiprogramming

Multiple programs have been retained in primary memory at the identical time in multiprogramming, and the CPU switches among them, ensuring that the Central Processing Unit is constantly executing a program. The OS starts by running one program from memory; if this application requires a delay, like an input/output activity, the OS switches to a different program. Multiprogramming makes the CPU work harder. Multiprogramming systems create an environment where in the different system resources are properly used, but they do not allow for the interaction of consumer with the computer (Christopher et al., 1993).

- Benefits:
  - Excessive central processing unit usage;
  - It looks that numerous programs are given central processing unit time virtually at the same time.
- Drawbacks:
  - There is a need for CPU scheduling;
  - Memory management is essential to support several jobs in memory.

7.7.5. Parallel System

It should be noted that the system contains greater than one CPU. Such processors communicate with one another through the computer bus, the clock, the memory, and the input/output devices (Plagemann et al., 2000).

The benefit is that capacity may be increased (the number of programs finished in unit time).

7.7.6. Distributed System

Spread the processing among many physical processors. It entails using a communication link to join two or more separate computer systems. As a result, every Central Processing Unit has its OS and local memory, and processors connect via a variety of communication channels, like higher-speed buses or landlines (Figure 7.13) (Kronenberg et al., 1986).

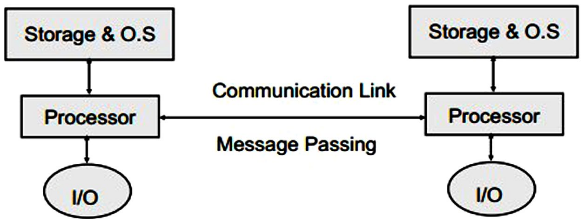


Figure 7.13. Diagram of distributed systems.

Source: https://www.researchgate.net/figure/The-block-diagram-of-the-distributed-system_fig5_4351563.

Benefits of distributed systems:

- **Sharing of Resources:** You may share printers and files.
- **Increased Calculation Speed:** A job may be divided so each processor may work on a portion of it at the same time, which is known as load sharing.
- **Reliability:** If one CPU fails, the other CPUs will continue to work normally.
- Electronic mail, ftp, and other forms of communication (Jo et al., 2014).

7.7.7. Personal Computer

Personal computers (PC) are computer systems that are only used by one person. Multi-user and multitasking capabilities were not available in personal computer OS (Corral et al., 2012). Rather than increasing input/output and CPU usage, the purpose of personal computer OS was to enhance user convenience and response. Apple Macintosh and Microsoft Windows are two examples.

7.8. OPERATING SYSTEM (OS) SERVICE

An OS is a software platform that delivers services to applications and their users. The OS provides the following services (Gligor, 1984):

1. **Input/Output Operation:** Input/output refers to every file or Input/output device. While operating, the software may require any Input/output device. As a result, the OS should be capable to deliver the necessary Input/output.
2. **Program Execution:** The OS executes and loads a program into memory. The software should be capable to terminate its operation in one of two ways: abnormally and normally.
3. **Communication:** For a while, data flow between two processes is necessary. Both procedures take place on the similar computer or separate machines linked by a computer network.
4. **Manipulation of File System:** The program must be able to write or read files. The OS allows the software to work with files.
5. **Detection of Errors:** Errors might arise in the CPU, input/output devices, or memory hardware. The OS must be continually aware of potential faults. It must take the necessary steps to guarantee that computing is accurate and reliable.
6. Communication can be accomplished in ways (Wentzlaff et al., 2009):

i. A common memory;
ii. Message transmission.

7.9. OPERATING SYSTEM (OS) OPERATIONS

A multi-user OS allows for more effective system operations (Peter et al., 2015):

- **Allocation of Resources:** For running the work at the same time.
- **Accountancy:** Account billing and use statistics are handled by accounting.
- **Security:** Make sure that accessibility to the resources of the system is restricted.

Interrupt-driven are the lifeblood of today's OS. An OS would sit quietly, waiting for anything to occur when there are no operations to run, no input/output devices to serve, and no consumers to reply to. The appearance of an interrupt or a trap nearly always signals the start of an occasion (Hansen, 1973). *A trap* is a software-generated interrupt that is triggered by either an error (such as division by zero or incorrect memory access) or a particular request from a user program to execute an OS service. Separate pieces of code in the OS determine what action must be taken for every sort of interrupt. An interrupt service procedure is included, which is in charge of dealing with the interruption.

Because the OS and users share the computer systems software and hardware sources, we must ensure that a bug in a user application only affects the one program that had been executing. Because of the sharing, a flaw in one application might harm multiple processes (Wentzlaff et al., 2009). *One OS should be constructed in such a way that an inaccurate (or malicious) application may not force other programs to run wrongly.*

7.9.1. Dual-Mode Operation

We need to be capable to tell the difference between OS code and consumer-specified code. The strategy is to split the 2 operating modes: *user mode and kernel mode* (known as privileged mode, system mode, or supervisor mode). *The mode bit* is known as a *mode bit* added to the computer's hardware that indicates the present mode: user (1) or kernel (0) (Saha et al., 2003). A dual-mode operation gives us the ability to defend the OS from rogue consumers, as well as rogue consumers from each other.

System calls allow consumer software to request that the OS do duties that are reserved for the OS on its behalf (Figure 7.14).

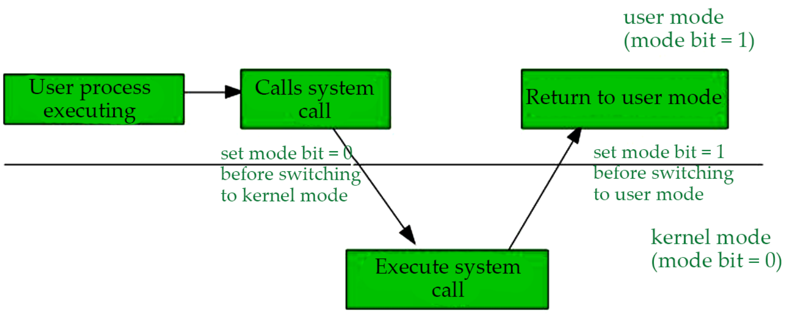


Figure 7.14. Dual mode operations in operating system.

Source: https://www.geeksforgeeks.org/dual-mode-operations-os/.

7.9.2. Protection CPU

The OS should have control over the CPU to make sure that it remains stable. To avoid a consumer application becoming caught in an infinite loop or failing to invoke system services and never surrendering control to the OS, we should block them from doing so (Zhang et al., 2011). We may utilize a timer to help us attain this aim. Using a timer, you may force the computer to shut down after a defined variable or fixed length of time.

7.10. OPERATING SYSTEM (OS) COMPONENTS

The elements of OS are discussed in subsections.

7.10.1. Process Management

In a multi-programming environment, the OS determines which processes receive processor time and for how long. The OS is in charge of the following procedure management activities (Fassino et al., 2002):

- Procedures that are paused and resumed;
- Both consumer and system procedures can be created and deleted;
- Providing techniques for dealing with deadlocks;
- Providing communication tools for processes;
- Providing synchronization methods for processes.

7.10.2. Memory Management

The main memory consists of a vast arrangement of words or bytes, each with its address. In terms of memory, the OS is in charge of the following tasks (Anderson et al., 1991):

- Monitoring of which bits of memory are being utilized and from whom at any given time;
- Determining which operations (or sections of operations) and data should be moved into and out of the memory;
- Assigning and reassigning memory space as required.

7.10.3. File System Management

The OS is responsible for the given below file management activities (Comer, 2011):

- File creation and deletion;
- Organizing files by creating and removing folders;
- Providing primitives for working with directories and files;
- File mapping to secondary storage;
- Using a stable (nonvolatile) storage medium to back up files.

7.10.4. Secondary Storage Management

OS In terms of disc management, the OS supports the following features (Chen et al., 1994):

- Controlling the amount of free space available;
- Allocation of storage;
- Scheduling on the hard drive.

7.10.5. System Call and System Program

System calls are used to connect a running program to the OS. The consumer may not perform privileged instructions; instead, the consumer must ask the OS to do so via system calls. Traps are used to implement system calls (Anderson et al., 1991).

Through the trap, the OS gets control, changes to kernel mode, conducts service, then switches back to user mode and returns control to the user.

The graphic shows an instance of how the OS uses system calls to read information from one file and transfer it to the next file. This amount of the data has never been seen by the programmer (Mullender et al., 1990):

Obtain the name of the input file

Write prompt on the screen

Accept input

Obtain output filename

Write prompt on the screen

Example system call sequence

Accept input

Open the input file

If the file does not exist, abort

Create output file

```
If the file exists, abort
Loop:  Read from input file
Write to the output file
Until reading fail
Close output file
Write completion message on the screen
Terminate normally
```

Consumers do not have to design their environment for program development (compilers, editors) or program execution (shells) since system programs give fundamental services.

7.10.6. Protection and Security

Protection A method that regulates the access of programs or consumers to both systems is referred to as protection. The security system should be able to (Wulf et al., 1974):

- Differentiate between authorized and unauthorized consumers;
- Specify the control that will be implemented;
- Establish a mechanism for enforcing the rules.

Security precautions are in charge of protecting a computer system from both external and internal threats.

REFERENCES

1. Agarwal, A., Hennessy, J., & Horowitz, M., (1988). Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems (TOCS), 6*(4), 393–431.
2. Anderson, T. E., Levy, H. M., Bershad, B. N., & Lazowska, E. D., (1991). The interaction of architecture and operating system design. *ACM SIGPLAN Notices, 26*(4), 108–120.
3. Austin, T., Larson, E., & Ernst, D., (2002). SimpleScalar: An infrastructure for computer system modeling. *Computer, 35*(2), 59–67.
4. Brown, K. A., & Mitchell, T. R., (1991). A comparison of just-in-time and batch manufacturing: The role of performance obstacles. *Academy of management Journal, 34*(4), 906–917.
5. Cardenas, A. F., (1973). Evaluation and selection of file organization—A model and system. *Communications of the ACM, 16*(9), 540–548.
6. Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., & Patterson, D. A., (1994). RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR), 26*(2), 145–185.
7. Christopher, W. A., Procter, S. J., & Anderson, T. E., (1993). The nachos instructional operating system. In: *USENIX Winter* (pp. 481–488).
8. Clark, R. K., Jensen, E. D., & Reynolds, F. D., (1992). An architectural overview of the Alpha real-time distributed kernel. In: *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures* (pp. 27–28).
9. Comer, D., (2011). *Operating System Design: The Xinu Approach, Linksys Version* (Vol. 1, pp. 3–9). Chapman and Hall/CRC.
10. Corral, L., Sillitti, A., & Succi, (2012). Mobile multiplatform development: An experiment for performance analysis. *Procedia Computer Science, 10*, 736–743.
11. Creasy, R. J., (1981). The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development, 25*(5), 483–490.
12. Dandamudi, S. P., (2003). *Fundamentals of Computer Organization and Design* (Vol. 7, p. 1–5). Berlin, Heidelberg: Springer.
13. Eager, B., & Lister, A., (2016). *Fundamentals of Operating Systems* (Vol. 1, pp. 1–10). Macmillan International Higher Education.

14. Estrin, G., (1960). Organization of computer systems: The fixed plus variable structure computer. In: *Papers Presented at the May 3–5, 1960, Western Joint IRE-AIEE-ACM Computer Conference, 5*(3), 33–40.
15. Fassino, J. P., Stefani, J. B., Lawall, J., & Muller, G., (2002). Think: A software framework for component-based operating system kernels. In*: 2002 USENIX Annual Technical Conference (USENIX ATC 02)*.
16. Gligor, V. D., (1984). A note on denial-of-service in operating systems. *IEEE Transactions on Software Engineering,* (3), 320–324.
17. Haber, R., & Unbehauen, H., (1990). Structure identification of nonlinear dynamic systems—a survey on input/output approaches. *Automatica, 26*(4), 651–677.
18. Hansen, P. B., (1973). *Operating System Principles* (3rd edn., p. 4–9). Prentice-Hall, Inc.
19. Hildebrand, D., (1992). An architectural overview of QNX. In: *USENIX Workshop on Microkernels and Other Kernel Architectures* (pp. 113–126).
20. Hughes, L., (2000). An applied approach to teaching the fundamentals of operating systems. *Computer Science Education, 10*(1), 1–23.
21. Jo, K., Kim, J., Kim, D., Jang, C., & Sunwoo, M., (2014). Development of autonomous car—Part I: Distributed system architecture and development process. *IEEE Transactions on Industrial Electronics, 61*(12), 7131–7140.
22. Kronenberg, N. P., Levy, H. M., & Strecker, W. D., (1986). VAXcluster: A closely-coupled distributed system. *ACM Transactions on Computer Systems (TOCS), 4*(2), 130–146.
23. Lange, J., Pedretti, K., Hudson, T., Dinda, P., Cui, Z., Xia, L., & Brightwell, R., (2010). Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (pp. 1–12). IEEE.
24. Litzkow, M., & Livny, M., (1990). Experience with the condor distributed batch system. In: *IEEE Workshop on Experimental Distributed Systems* (pp. 97–101).
25. Mullender, S. J., Van, R. G., Tananbaum, A. S., Van, R. R., & Van, S. H., (1990). Amoeba: A distributed operating system for the 1990s. *Computer, 23*(5), 44–53.

26. Murphy, D. L. (1972). Storage organization and management in TENEX. In: *Proceedings of the December 5–7, 1972, Fall Joint Computer Conference, Part I* (pp. 23–32).
27. Musina, O., Putnik, P., Koubaa, M., Barba, F. J., Greiner, R., Granato, D., & Roohinejad, S., (2017). Application of modern computer algebra systems in food formulations and development: A case study. *Trends in Food Science & Technology, 64*, 48–59.
28. Peter, S., Li, J., Zhang, I., Ports, D. R., Woos, D., Krishnamurthy, A., & Roscoe, T., (2015). Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS), 33*(4), 1–30.
29. Plagemann, T., Goebel, V., Halvorsen, P., & Anshus, O., (2000). Operating system support for multimedia systems. *Computer Communications, 23*(3), 267–289.
30. Ritchie, D. M., & Thompson, K., (1978). The UNIX time-sharing system. *Bell System Technical Journal, 57*(6), 1905–1929.
31. Ritchie, D. M., (1984). The UNIX system: A stream input-output system. *AT&T Bell Laboratories Technical Journal, 63*(8), 1897–1910.
32. Robey, D., (1981). Computer information systems and organization structure. *Communications of the ACM, 24*(10), 674–687.
33. Rosenblum, M., & Ousterhout, J. K., (1992). The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS), 10*(1), 26–52.
34. Saha, B. B., Koyama, S., Kashiwagi, T., Akisawa, A., Ng, K. C., & Chua, H. T., (2003). Waste heat driven dual-mode, multi-stage, multi-bed regenerative adsorption system. *International Journal of Refrigeration, 26*(7), 749–757.
35. Seltzer, M., & Small, C., (1997). Self-monitoring and self-adapting operating systems. In: *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)* (pp. 124–129). IEEE.
36. Silberschatz, A., Peterson, J. L., & Galvin, P. B., (1991). *Operating System Concepts* (Vol. 10, No. 3, pp. 2–8). Addison-Wesley Longman Publishing Co., Inc.
37. Stallings & W. (2003). Computer organization and architecture: Designing for performance. *Pearson Education India, 25*(4), 16–26.
38. Wentzlaff, D., & Agarwal, A., (2009). Factored operating systems (fos) the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review, 43*(2), 76–85.

39. Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., & Pollack, F., (1974). Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM, 17*(6), 337–345.
40. Zhang, F., Chen, J., Chen, H., & Zang, B., (2011). Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Vol. 23, pp. 203–216).

CHAPTER 8

TIMELINE OF COMPUTER WINDOWS AND ITS FEATURES

CONTENTS

8.1. INTRODUCTION

Whatever comes to mind when you consider the history of Windows? Logos that are instantly recognizable? Changing the Start menu's appearance? The advent of Live Tiles? All of this and much more is included in the past of Microsoft's main operating system (OS). Windows has seen several reincarnations during the last 35 years (Rushinek and Rushinek, 1997). Here we examine 14 different Windows version because they all represent significant milestones in the history of computers (Inglot and Liu, 2014; Rajesh et al., 2015). Before we get in to Windows history, it is worth considering the status of computing prior to Windows.

8.2. MS-DOS AND WHAT CAME BEFORE

Windows may appear to have been around for a long time, nonetheless it hasn't. Microsoft's original OS was not Windows. In reality, before Windows, PCs ran on a different OS called MS-DOS. Despite as the initial edition of Windows, browsing your PC with MS-DOS took time, needed guide text command input, and no permit for multitasking (the capability to run multiple programs at once) (Laric, 1995; Akbal et al., 2016).

At minimum in 1985, Windows wasn't so many a fresh OS as this was a response to the problems which an OS like MS-DOS brought. Windows 1.0 was designed as a graphical user interface (GUI) to run on top of MS-DOS, making it simpler to browse PCs running MS-DOS – it's simpler to focus on a screen and press an icon to start a program than it is to write numerous instructions to do the same activity.

However, Windows wasn't the primary GUI to address concerns such as needing to browse through text instructions. Apple and Xerox were the first two firms to get there. Apple produced "the first marketable computers with a GUI" in 1983, according to Wired. It was dubbed the Lisa. The Lisa was the very first commercialized computer with a GUI, although it wasn't the oldest computer with a GUI ever. Xerox released the first one in 1981, and that it was named as the Star (Huxford, 1993; Ma et al., 2002).

Despite being three or four years later to the GUI party, Microsoft was capable to sell its initial windows version at a far lower cost than its competitors, providing it a substantial advantage. The development of Windows is discussed in further sections.

8.3. WINDOWS 1.0

Windows 1.0 emerged in 1985 as a GUI to be utilized in combination with MS-DOS. The implementation of Windows 1.0 as a GUI ensured that MS-DOS users no longer had to physically type text instructions only to execute simple operations. They could now do jobs and peruse their own files simply by picking and pressing on menus. The ionic Windows 1.0 cost $99 at the time of its debut and presented numerous computer users to pull-down icons, menus, and conversation boxes per Microsoft. It also had the capacity to multifunction apps and "transfer data across programs," which was a first for a Microsoft OS (Figure 8.1) (Harris, 1999; Hamadani et al., 2011).

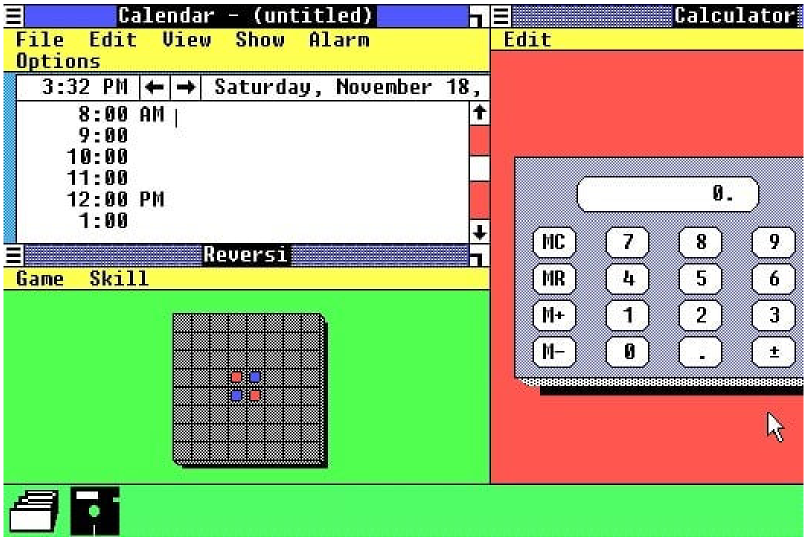


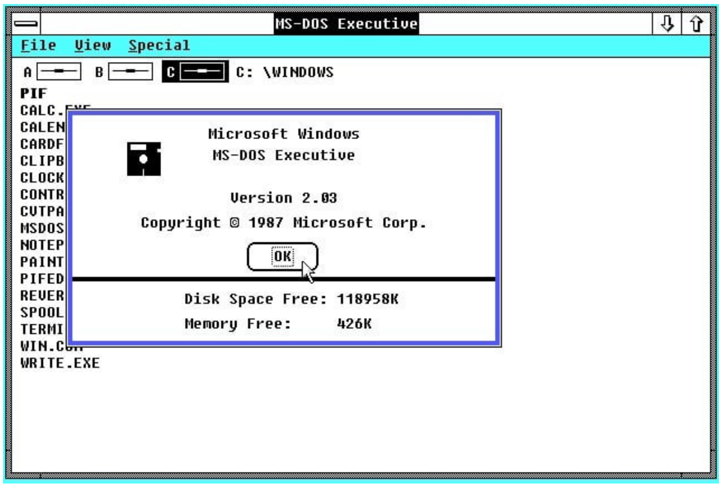
Figure 8.1. Windows 1.0 image.

Source: https://winworldpc.com/product/windows-10/101.

Don't be fooled by Windows 1.0's bare-bones appearance: the OS also had Windows Paint, Windows Write, a calendar, a clock, a notepad, a cardfile, a file manager, a terminal app, as well as a game named Reversi, according to The Verge.

8.4. WINDOWS 2.0

It wasn't much till Microsoft launched a follow-up to its first GUI-enhanced OS. Windows 2.0 was introduced after two years in 1987, by the technology business. This version of Windows contained noteworthy features such as overlapped windows, resizable windows, shortcut keys, and VGA graphics capabilities. Word and Excel's first Windows versions were also released with Windows 2.0 (Figure 8.2) (Archibugi and Pietrobelli, 2003; Lin and Vincent, 2012).



```
Microsoft Windows
MS-DOS Executive

Version 2.03
Copyright © 1987 Microsoft Corp.

Disk Space Free: 118958K
Memory Free:     426K
```

Figure 8.2. Windows 2.0 image.

Source: https://winworldpc.com/product/windows-20/20.

8.5. WINDOWS 3.0

Microsoft's next great achievement was the launch of Windows 3.0. This windows version is largely regarded as the beginning of Windows' international prominence as a desktop OS. In 1990, Microsoft released Windows 3.0, which supported 256 colors. More crucially, as PCMag points out, it had "multitasking DOS apps," that may have led to Windows' popularity boom. An additional important aspect of Windows 3.0 is, this is the form in which the famous desktop game Solitaire initially appeared (Figure 8.3) (Smith, 1996; Uzunboylu et al., 2011).

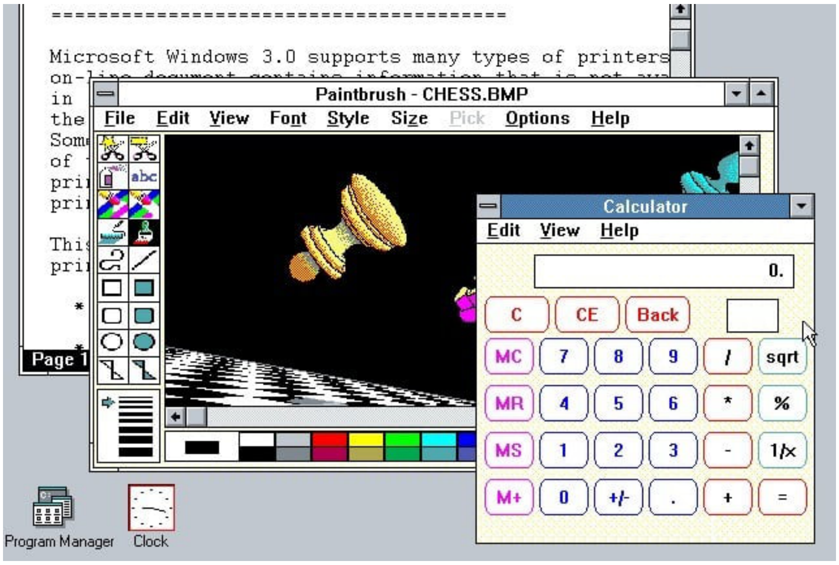
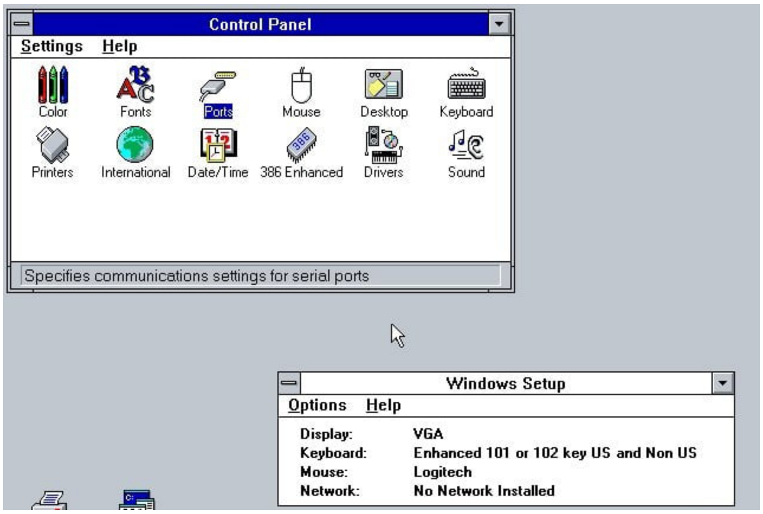


Figure 8.3. Windows 3.0 image.

Source: https://winworldpc.com/product/windows-3/30.

8.6. WINDOWS 3.1

Two years later, some other new version of OS occurred, this time bringing Windows to its best-known versions, 3.1. Although the fraction in its designation implies that it was a slight improvement to 3.0, but it was not. Rather than that, Windows 3.1 introduced a number of critical new features in 1992, including compatibility for TrueType fonts, a capability to drag and drop icons, and capability for OLE composite files (files that join components from various plans). Additionally, A Guardian reports that it is the first variety of Windows to be issued on CD-ROM (Figure 8.4) (Scoville, 1993; Lee et al., 2011).



Specifies communications settings for serial ports

Display:	VGA
Keyboard:	Enhanced 101 or 102 key US and Non US
Mouse:	Logitech
Network:	No Network Installed

Figure 8.4. Windows 3.1 image.

Source: https://winworldpc.com/product/windows-3/31.

8.7. WINDOWS 95

You're definitely thinking about Windows 95 when you consider the most famous version of Windows. That is due to that it was like a radical leaving artistically from earlier forms of Windows, and it set a quality for what we've arise to know from Windows OS. Windows 95 was released in 1995, as its name indicates. It was the oldest 32-bit version of Windows (prior editions had been 16-bit), and this included a number of new characteristics that would go down in history (Campbell, 1991; Nolze and Kraus, 1998). The taskbar, the Start menu, lengthy file names, and plug-and-play features are among them (at which marginal gadgets only required to be linked to a PC for work correctly). Internet Explorer, Microsoft's web browser, was also introduced with Windows 95 (Figure 8.5) (Campbell, 1992).

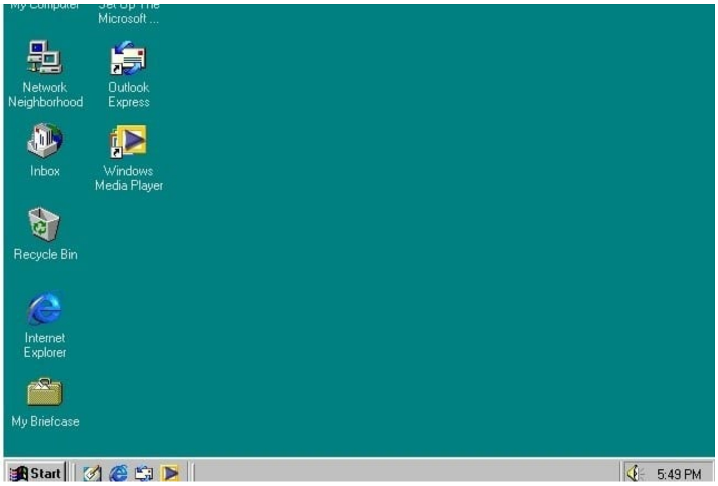


Figure 8.5. Image of Windows 95.

Source: https://microsoft.fandom.com/wiki/Windows_95.

Another noteworthy function? However, Windows 95 still operated with MS-DOS, dissimilar its forerunner, it didn't have to delay for the computer to boot into DOS first, as PCMag points out. This was the first moment Windows was permitted to boot straight from the hard drive (Barney, 1994).

8.8. WINDOWS 98

This is the Windows version with a title that corresponds to year this was introduced. If Windows 95 (finally) gave us Internet Explorer, Windows 98 tightened the web browser's hold on Microsoft's OS. This form of Windows not one included Internet Explorer 4.01, nonetheless also a bevy of additional internet-related apps and features, like Microsoft Chat, the Web Publishing Wizard. and Microsoft Outlook (Figure 8.6).

Figure 8.6. Screenshot of Windows 98.

Source: https://microsoft.fandom.com/wiki/Windows_98.

Windows 98 also included expanded compatibility for USB drives and Macromedia apps (Shockwave and Flash).

8.9. WINDOWS 2000

Windows 2000 placed a strong emphasis on accessible, introducing plenty of new features to the OS, like StickyKeys, an elevated design, Microsoft Magnifier, an on-screen keyboard, and Microsoft Narrator, a screen reader (Figure 8.7) (Ho et al., 2001; Pfeiffer et al., 2003).

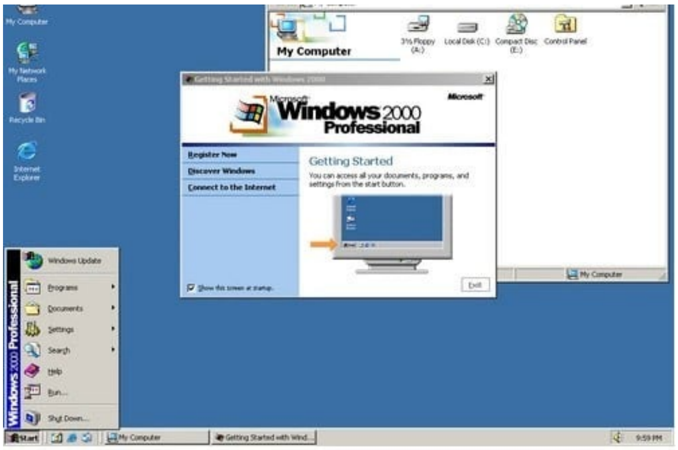


Figure 8.7. Image of Windows 2000.

Source: https://microsoft.fandom.com/wiki/Windows_2000.

Additionally, Windows 2000 had a Multilingual User Interface that let users to select the languages in which their presentation will be shown. Users of Windows 2000 have a range of language options, such as Arabic, Japanese, and Greek (Guo et al., 2010; Prentice et al., 2013).

8.10. WINDOWS ME

"ME" stands for "Millennium Edition" in Windows ME. It also was called "The Mistake Edition," which was a less favorable title. When Windows ME was released in 2000, it was given the moniker because "customers had issues downloading it, enabling it to start, getting it to operate with other software or hardware, and having it to quit operating" (Figure 8.8).

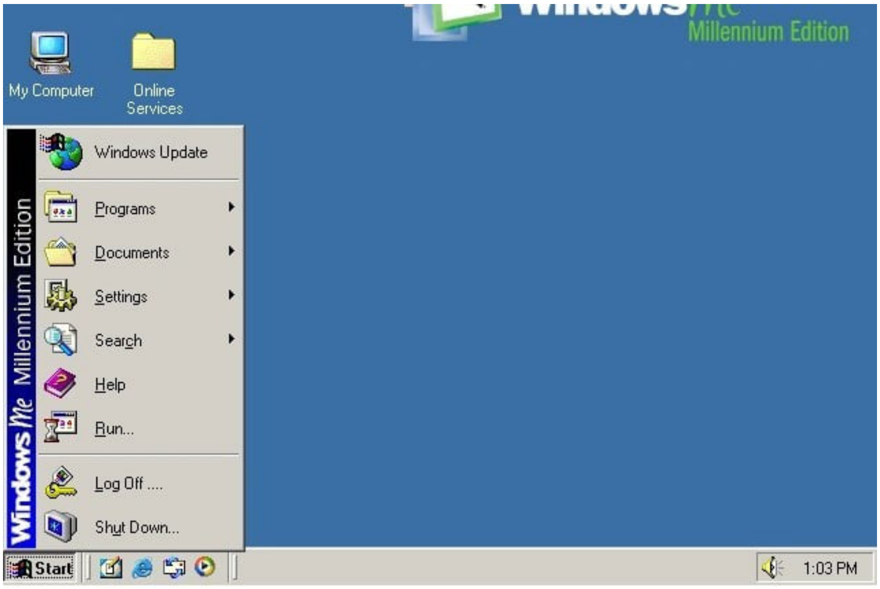


Figure 8.8. Image of Windows ME.

Source: https://microsoft.fandom.com/wiki/Windows_Me.

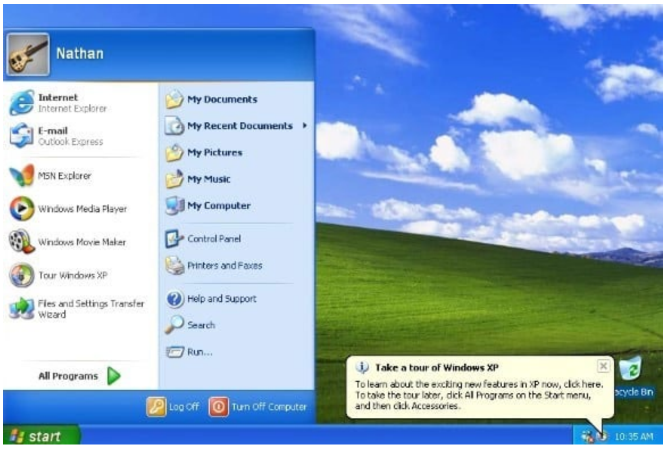
Even after its terrible start, this still handled in order to provide us with a valuable device (Chau and Hui, 1998). System restores, a recovery characteristic which, if your computer is turned up having issues because of poorly executed installation of a program or upgrade, can eliminate some these updated information and rebuild your computer to the way it was formerly the infringing update screwed with it. System Restore, in classic

Mistake Edition manner, had its own challenges to work out before becoming genuinely fantastic. For example, it occasionally messed up the restoration by reinstalling items such as viruses that had already been deleted (Dong, 1999; Zhang et al., 2016).

8.11. WINDOWS XP

Windows XP was introduced in 2001 and is largely regarded as one of Microsoft's best Windows OSs. The OS was available in two versions: Home and Professional. However, the Professional was designed for usage in business environments, while Home was designed for personal use. Share of XP's success, according to TechRadar, may be ascribed to the fact that it was released at the same time as a surge in PC sales, thus for numerous new operators, "Windows XP was what arrived with their oldest computer."

The popularity of XP may be followed back to the OS itself. And besides, since it lasted 13 years till Microsoft withdrew support for it in 2014, there must have been something appealing about its design. Because it is actually meant to be consumer-friendly, it has achieved some commercial success. Bright colors, a bright green Start button, and configurable theme tune were lastly included with this windows version, giving it a warm and appealing design. It also included additional features including as native CD ripping software, desktop search, remote desktop, and (soon) enhanced security (Figure 8.9) (Sullivan, 1996; Dong, 1999).



Take a tour of Windows XP
To learn about the exciting new features in XP now, click here. To take the tour later, click All Programs on the Start menu, and then click Accessories.

Figure 8.9. Windows XP image.

Source: https://microsoft.fandom.com/wiki/Windows_XP.

8.12. WINDOWS VISTA

Regrettably, Vista was yet another critically derided windows edition. When Vista was introduced in 2007, one of the primary criticisms was which its afresh developed design interface (dubbed Aero Glass) did not always work very well along first hardware or specific graphics drivers on newer PCs. Other critiques leveled against Vista were its poor presentation, exorbitant pricing, excessive system resource consumption, and, whereas the User Account Control function-maintained users secure, the continuous dialog boxes presented by the program were vexing (Figure 8.10) (Uslan and Su, 1997; Cota-Robles and Held, 1999).

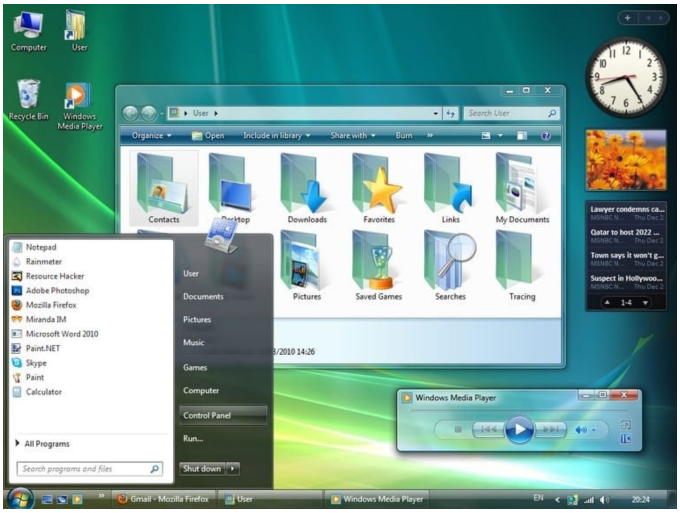


Figure 8.10. Screenshot of Windows Vista.

Source: https://www.digitaltrends.com/computing/the-history-of-windows/.

Vista attempted to do too much, too quickly, and was burnt as a result. Although it included numerous valuable functions, such as DirectX 10, Windows Defender, (for PC gaming), Windows DVD Maker, and speech recognition, it was not without flaws (Gratze et al., 1998).

8.13. WINDOWS 7

Microsoft released Windows 7, a new windows version, 2 years later. Microsoft needed to develop for Vista's shortcomings, and Windows 7 did exactly that. Windows 7 is more simplified than Vista, and it essentially removes numerous functionalities from preceding windows versions, especially Vista. In truth, Microsoft did not have at least 4 Vista apps in Windows 7: Windows Movie Maker, Windows Photo Gallery, Windows Mail and Windows Calendar (Lio and Nghiem, 2004).

Handwriting recognition, improved overall speed, interactive thumbnails preview for reduced program windows, a desktop slideshow function, Windows Media Player 12, and Internet Explorer 9, were all included in Windows 7 (Figure 8.11) (Bolosky et al., 2000).

Figure 8.11. Windows 7 image.

Source: https://www.digitaltrends.com/computing/the-history-of-windows/.

8.14. WINDOWS 8

Visually, Windows 8 was a sea change from its forerunners. It's time to discuss the tiled display screen. The Start screen had slates dubbed Live Tiles that served as dynamic app shortcuts, allowing you to start your apps while simultaneously displaying mini-updates about them (like the quantity of unread messages). The Start screen was intended for replacing the Start menu. In this configuration, Windows 8 retains the classic Windows desktop, which is where applications are executed (Figure 8.12) (Westerlund and Danielsson, 2001; Naik, 2004).

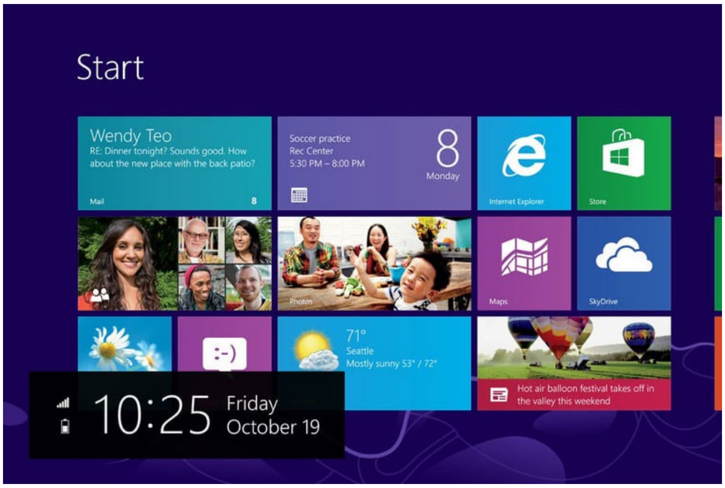


Figure 8.12. Screenshot of Windows 8.

Source: https://news.microsoft.com/accessing-system-commands/.

While not everyone was delighted with Windows 8's tablet-centric redesign, it provided several more features like the option to USB 3.0 connectivity, login with a Microsoft account, a real lock screen (visually comparable to a phone home screen), and Xbox Live integration (Warner, 2001; Swift et al., 2002).

8.15. WINDOWS 8.1

Consumers were not pleased with the startling Windows 8 Start screen and the disappearance of the Start menu. In response, Microsoft introduced Windows 8.1 as a free update to report customer concerns about its predecessor (Figure 8.13) (Sechrest and Fortin, 2001; Ganapathi et al., 2006).

Figure 8.13. Windows 8.1 image.

Source: https://news.microsoft.com/windows-8-1-preview-lock-screen/.

Microsoft made certain changes in Windows 8.1, such as adding a real Start button to the toolbar and allowing users to get the desktop immediately afterward signing in (in its place of actuality received by the dreaded Start screen). Microsoft didn't waste any time in releasing this patched-up version of Windows: Windows 8 was introduced in 2012, followed by Windows 8.1 in 2013.

8.16. WINDOWS 10

Windows 10 was released in 2015 and is the latest version of Microsoft's OS. Once it launched, it was clear that Microsoft sought to improve its usage of Live Tiles instead of completely eliminate them. It harmed the following in Windows 10: It replaced Windows 8's hated Start screen with a wider Start menu that makes usage of Live Tiles and other types of program icons. It was successful (Bickel et al., 2002; Stiegler et al., 2006).

Additionally, according to the Verge, the 2015 edition of Windows 10 introduced Cortana, a native digital personal helper; the capability to convert among tablet and desktop modes; and a new online browser (Microsoft Edge).

Since its introduction in 2015, Windows 10 has also gotten quite frequent upgrades. They are referred to as feature updates and occur each six months. They are always accessible for free via Windows Update. Indeed, the following function is not that far away: Windows 10 20H1 is scheduled for release in the spring of 2020, maybe around May. This update is likely to bring a revamped Cortana experience and the capability to restore Windows "simply selecting the choice to Cloud downloading Windows, in the absence of having to produce installation discs" (Figure 8.14).

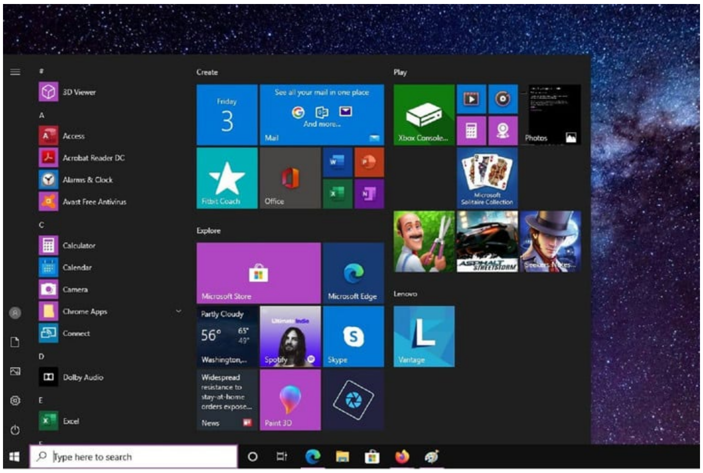


Figure 8.14. Windows 10 image.

Source: https://www.digitaltrends.com/computing/the-history-of-windows/.

8.17. WINDOWS 11

Windows always has been to serve as a platform for global innovation. It has served as the backbone of multinational enterprises and as a platform for scrappy initiatives to become household names. Windows gave birth to and raised the web. It's where most of us sent our first mail, started playing our first PC game, and coded our first line. Windows is the platform on which over a trillion people these days rely to create, connect, learn, and succeed.

We don't take the responsibility of creating for several individuals casually. We moved from adapting the PC into our living to irritating to integrate our entire life into the PC over the last 18 months, which has resulted in an extraordinary shift in the way, we use our PCs. Our gadgets were not just where we went for conferences, classes, and to get tasks completed; they were also where we went out and played games with mates, obsessively watch our favorite programs, and, maybe most importantly, communicate with each other. We ended up digitally reproducing the workplace conversation, hallway banter, exercises, happy hours, and holiday festivities (Bird et al., 2009; Zimmermann et al., 2010).

The transition in PC that we saw and felt was quite profound — from somewhat utilitarian and useful towards something emotional and personal. It is what motivated us to create the next iteration of Windows. To offer a familiar environment in which you may create, study, play, and, most highly, interact in novel ways (Whitehouse, 2007).

Nowadays, I am honored and pleased to present you Windows 11, the OS to bring you nearer to the things you care about.

8.17.1. Redesigned for Productivity, Creativity, and Ease

We've optimized experiences of user and design in order to boost your production and stimulate your originality. This is contemporary, bright, spotless, and lovely. Everything, from the fresh Start sign and toolbar to the sounds, fonts, as well as icons, was designed with the aim of tapping you in command and instilling a sense of comfort and serenity (Figure 8.15) (Hargreaves et al., 2008; Narayan et al., 2009).

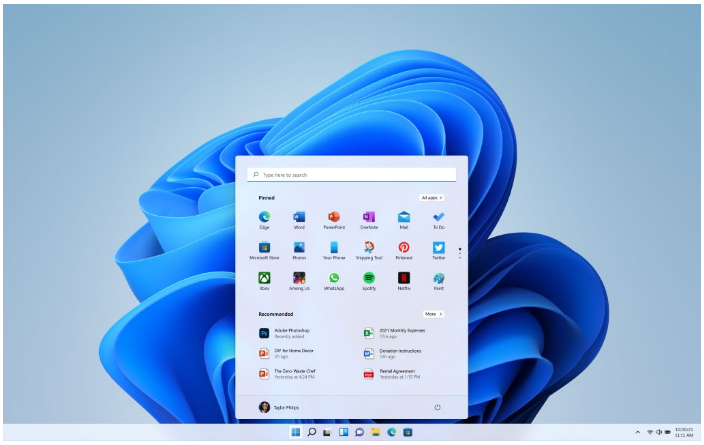


Figure 8.15. Perfect interface design.

Source: https://blogs.windows.com/windowsexperience/2021/06/24/introducing-windows-11/.

We centered Start and made it humbler to immediately locate what you're observing for. Begin leverages the web and Microsoft 365 to show your new files irrespective of the stage or device on which you were reading them formerly, even if it was an Android or iOS gadget.

Windows is all about allowing you to work perfectly as you choose, with many windows and the capability to link programs together. Snap Groups, Desktops, and Snap Layouts, are all new in Windows 11 and give another more adaptive approach to multitasking and keep at the top of just what people require to get accomplished. These are new tools that will help you manage your windows and maximize your screen display so you can view just what you want from a visually appealing layout. You may also build various Desktops for different aspects of your life and configure them to your preferences — for example, a Desktop for working, gaming, and education (Figures 8.16 and 8.17) (Ray and Schultz, 2007; Li et al., 2008).

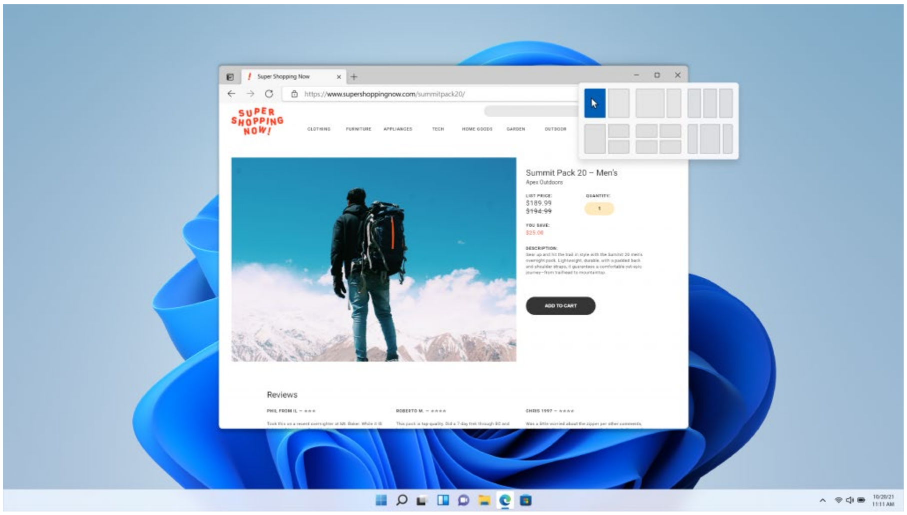


Figure 8.16. Multiple window flexibility.

Source: https://www.techrepublic.com/article/windows-11-cheat-sheet-everything-you-need-to-know/.

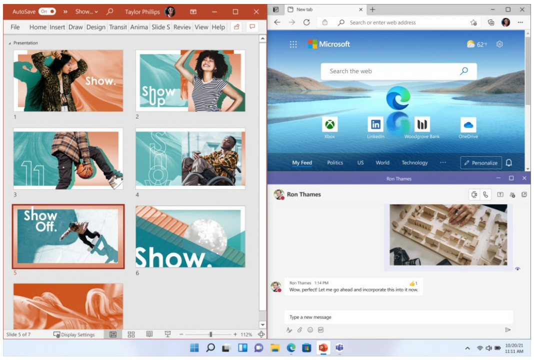


Figure 8.17. Windows 11 removes the complexity and replaces it with simplicity.

Source: https://blogs.windows.com/windowsexperience/2021/06/24/introducing-windows-11/.

8.17.2. Fast Connectivity

Another crucial component of putting you closer to your passion is getting you nearer to your loved ones. The previous 18 months have influenced how we develop meaningful digital interactions with individuals. Even as we begin to move to more personal engagement, we want to keep it simple for individuals to stay connected regardless of their location. And we don't want your device or platform to be a hindrance (Figure 8.18) (Narayan et al., 2009).

Figure 8.18. A more efficient method of communicating with the individuals you care about.

Source: https://www.techrepublic.com/article/windows-11-cheat-sheet-everything-you-need-to-know/.

We're delighted to debut Chat from Microsoft Teams embedded into the toolbar in Windows 11. Today, you can link directly with most of your in-person contacts by text, chat, audio, or video, regardless of their device or platform, Android, iOS or on Windows. If the individual on the other end does not have the Teams app, you still can communicate with them using two-way SMS (Purcell and Lang, 2008; Zhang et al., 2010).

Additionally, Windows 11 provides a more usual approach to engage with friends and family via Teams, letting you to rapidly mute and unmute a conversation or begin displaying right from the toolbar.

8.17.3. Perfect for Gaming

If you want to play video games, Windows 11 is the software platform for you. Gaming has always been essential to the Windows philosophy. A lots people across the globe now play the game on Windows to have fun and engage with their friends and family. Windows 11 makes full use of your system's capabilities, putting cutting-edge gaming technologies to work for you. DirectX 12 Ultimate allows spectacular, comprehensive visuals at high speeds; DirectStorage enables fast loading and more realistic gaming environments; and Auto HDR enables a larger, more varied spectrum of colors for a really intriguing visual experience. Our dedication to device support has not changed – Windows 11 supports all of your preferred PC gaming fittings and peripherals. With Xbox Game Pass for Desktop or Ultimate, players get access to hundreds of high-quality PC games, with new titles published on a constant schedule, and that is just as simple to discovery people to play with, whether on a PC or a console (Figure 8.19) (Thomas et al., 2013; Eterovic-Soric et al., 2017).

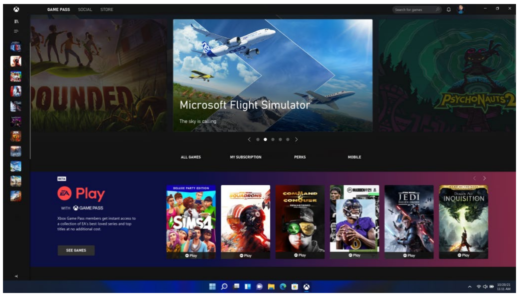


Figure 8.19. Offering the best possible PC gaming experiences.

Source: https://blogs.windows.com/windowsexperience/2021/06/24/introducing-windows-11/.

8.17.4. Faster to Get Information

Widgets, a new personalized feed powered by AI and Microsoft Edge's best online performance, bring you closer to the information and news you care about faster in Windows 11. Even when we're at our most focused and creative, we need pauses to check in with the outside world or recharge our brains. These days, we constantly check our phones for news, weather, and notifications. Your PC may now provide a similarly personalized experience. When you open your personalized feed, it slips over your screen like such a pane of glass, enabling you to continue to work uninterrupted. Widgets also provide producers and publishers more space inside Windows to deliver customized content. Our objective is to create a vibrant pipeline that benefits both customers and artists for both major companies and local creators (Figure 8.20) (Lallie and Briggs, 2011; Chien et al., 2014).

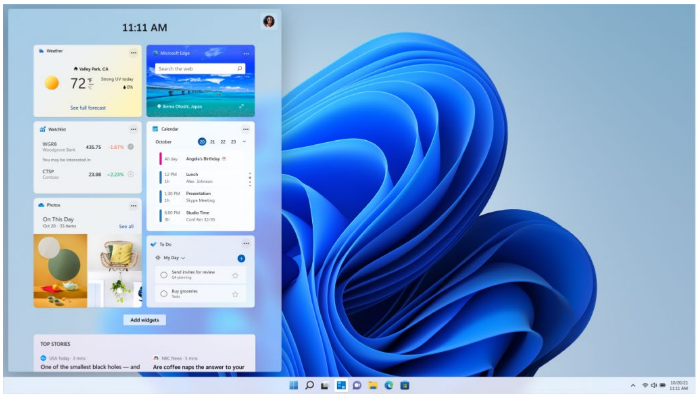


Figure 8.20. Obtaining knowledge in a more expedient manner.

Source: https://www.techrepublic.com/article/windows-11-cheat-sheet-everything-you-need-to-know/.

8.17.5. Microsoft Store

The latest Windows Store provides access to programs and content for viewing, producing, gaming, studying, and learning. It has been optimized for performance and boasts a completely new interface that is both simple to use and lovely to look at. Not only will we supply you with more apps than previous, but we'll also offer all content — apps, games, television shows, and movies – easier to locate and discover via curated stories and categories. We're excited to announce the upcoming addition of a number of premier first- and third-party apps to the Microsoft Store, such as Microsoft Teams, Visual Studio, Disney+, Adobe Creative Cloud, Zoom, and Canva – which all deliver fantastic experiences that entertain, inspire, and connect you. When you download an app from the App Store, you could be certain that it has already been properly vetted for security and family safety (Figure 8.21) (Talebi et al., 2012).

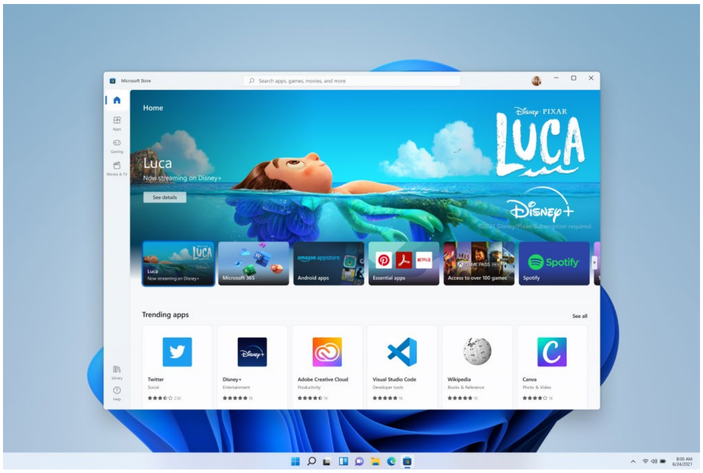


Figure 8.21. A latest Microsoft Store which combines your favorite programs and entertainment together in one place.

Source: https://blogs.windows.com/windowsexperience/2021/06/24/introducing-windows-11/.

8.17.6. Android Apps on PC

We're also excited to announce that for the first time we'll be introducing Android apps to Windows. Beginning later this year, customers will be able to find Android applications in the Microsoft Store and install them from the Amazon Appstore – picture shooting and publishing a TikTok video or utilizing Khan Academy Children for virtual learning directly from your PC. In the next months, we'll have more to say about this event. We are excited about our collaboration with Amazon and Intel, which will make use of Intel Bridge technology (Figure 8.22) (Odell and Chandrasekaran, 2012; Corregedor and Von Solms, 2013).

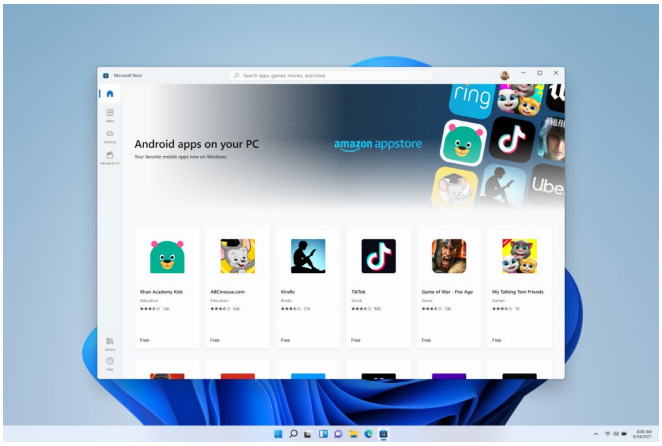


Figure 8.22. Android applications on a PC.

Source: https://www.pcmag.com/how-to/how-to-run-android-apps-in-windows-11.

8.17.7. Creating a More Open Ecosystem Unlocking New Opportunity for Developers and Creators

We are taking steps to increase the openness of the Microsoft Store in order to create additional commercial opportunities for creators and programmers. We're enabling developers and independent software vendors (ISVs) to bring their apps to the platform, regardless of whether they're developed as a Win32, universal windows app (UWP), or progressive web app (PWA), any other app framework, therefore boosting their reach and engagement. Furthermore, we're launching a progressive change to our revenue sharing practices, enabling developers to now bring their own commerce to our Store and keep 100% of the proceeds – Microsoft takes no share. With a competitive 85/15 revenue split, app developers may continue to use our commerce. We believe that promoting a more open environment helps our customers in the long term by giving them with secure, frictionless access to the apps, games, movies, programs, and online content that they want and need (Khatri, 2015).

8.17.8. Faster, More Secure and Familiar for IT

For IT professionals, Windows 11 is based on the same stable, well-matched, and recognizable Windows 10 foundation. You'll propose, make, and install

Windows 11 in the same way that you do now with Windows 10. Updating to Windows 11 will be similar to updating to Windows 10. As you incorporate Windows 11 into your estate, the same organization practices you have nowadays – such as Microsoft Endpoint Manager, cloud setup, Windows Update for Business, and Autopilot – will support your future environment. We are dedicated to app availability, which is a major design principle of Windows 11, just as we were with Windows 10. With App Assure, a service to help clients with 150 or more users address any app difficulties they may have at no additional cost, we stand by our guarantee that your apps will operate on Windows 11 (Mehreen and Aslam, 2015).

Windows 11 is also safe by default, with newly constructed security mechanisms that provide security from the chips to the cloud while allowing for increased efficiency and new experiences. To safeguard data and access across devices, Windows 11 has a Zero Trust-ready OS. We've worked very closely with our OEM and silicon suppliers to enhance security baselines in response to the changing threat landscape and the emerging hybrid work environment (Venčkauskas et al., 2015).

The Microsoft 365 blog has further information on Windows 11 as a computer system for mixed work and learning.

8.17.9. It Is a Great Time to Buy a PC

We've been working very closely with our device and semiconductor partners since the beginning of Windows 11 development to ensure seamless integration of software and hardware. That co-engineering starts with silicon innovation. From AMD and the extraordinary graphical depth provided by Ryzen processors to Intel's 11th generation and Evo Processors, to Qualcomm's AI capabilities, 5G, and Arm support, the creativity of our silicon suppliers ties together all the best of Windows 11 with the world's greatest hardware ecosystem.

And, in collaboration with Dell, HP, Lenovo, Samsung, Surface, and others, we've worked to guarantee that most PCs* available today are ready for Windows 11 – across a range of form factors and price ranges.

We've worked together to optimize Windows 11 not only for speed and accuracy, but also to take benefit of new touch, inking, and voice interactions.

When using Windows 11 on a laptop without even a keyboard, we've improved the touch experience by introducing more space between taskbar icons, larger touch targets, and subtle visual clues to enable resizing and dragging windows simpler, and also gestures. We're also allowing haptics to make utilizing your pen much more engaging and immersive, letting you to feel and hear the sensations as you click, edit, or doodle. Finally, we've made improvements to voice typing. Windows 11 recognizes what you say really well; it can dynamically capitalize for you and has voice commands. This is a terrific option for whenever you want to avoid typing and instead voice your thoughts (Hofmeester and Wolfe, 2012; Gao et al., 2013).

Beginning this Christmas season, Windows 11 will be offered as a free update for qualified Windows 10 PCs and for new PCs. Visit Windows.com and download the PC Health Check app to see if your existing Windows 10 PC is capable for the free upgrading to Windows 11. We're also collaborating with our store teams to ensure that Windows 10 PCs purchased today are prepared for the Windows 11 upgrade. The free upgrade will begin rolling out to compatible Windows 10 PCs this Christmas season and will continue until 2022. And, starting next week, we'll start sharing an early copy of Windows 11 with the Windows Insider Program — this is a dedicated group of Windows lovers whose feedback we value.

8.18. THE FUTURE OF WINDOWS

We're not saying Windows 11 will never come, and it's been five years since Windows 10 launched, and Microsoft is content with putting out new feature upgrades every six months for the current version of its OS. Plus, it's not as though such feature upgrades deprive Windows 10 customers of new functionality and design improvements. They occur two times a year and frequently include a huge list of issue patches, new toolkit, and cosmetic tweaks to the game's design — especially if they really do contain an unusual issue (Singh and Singh, 2017).

Even if Windows 11 does not materialize, this does not imply that Windows' long history of adaptation and creativity must come to a stop. Windows, particularly in current years, has evolved into something more than a desktop OS. Consider Windows Core OS. The Windows OS brand's future may lay with Core OS, which would be planned to be a stand-alone OS (just not an upgrade to Windows 10). Core OS is projected to become the flagship OS for smaller devices like as phones, tablets, and Chromebook such as laptops, having distinct Core OS versions for each kind of device. It is likely that the evolution of Windows will simply include the development of distinct (but it's still linked) OSs to meet the demands of an increasingly mobile society (Singh and Singh, 2016).

8.19. MAIN FEATURES OF MICROSOFT WINDOWS

Microsoft Windows comes with a variety of tools and programs to help you get through your computer and Windows. Click a link below to read much more about capabilities featured with Microsoft Windows.

8.19.1. Control Panel

The Control Panel is a set of tools that will assist you in configuring and manage services on your desktop. You can modify printer, video, audio, mouse, keyboard, time, and date, user profiles, installed apps, network connections, energy saver choices, and other settings.

The Control Panel in Windows 10 is found in the Menu bar, under Windows System. The Control Panel can also be launched from the Run box. Enter control by pressing Windows key + R. Alternatively, you may use the Windows key, type Control Center, and then hit Enter. Many Control Panel options are also available in the Windows 10 Options menu (Figure 8.23).

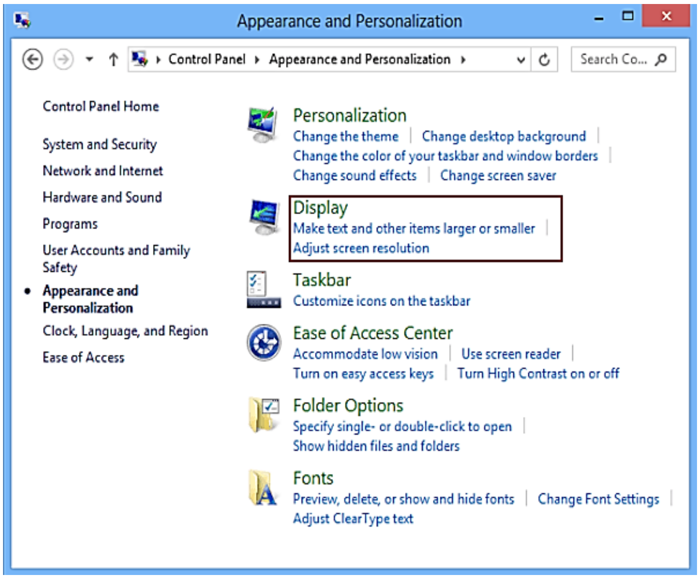

Appearance and Personalization
Control Panel ▸ Appearance and Personalization ▸
Search Co...
Control Panel Home
System and Security
Network and Internet
Hardware and Sound
Programs
User Accounts and Family Safety
Appearance and Personalization
Clock, Language, and Region
Ease of Access
Personalization
Change the theme | Change desktop background | Change the color of your taskbar and window borders | Change sound effects | Change screen saver
Display
Make text and other items larger or smaller | Adjust screen resolution
Taskbar
Customize icons on the taskbar
Ease of Access Center
Accommodate low vision | Use screen reader | Turn on easy access keys | Turn High Contrast on or off
Folder Options
Specify single- or double-click to open | Show hidden files and folders
Fonts
Preview, delete, or show and hide fonts | Change Font Settings | Adjust ClearType text

Figure 8.23. Display of the control panel.

Source: https://answers.microsoft.com/en-us/windows/forum/all/where-is-display-control-panel-in-windows-build/ce8fcc12-f3c2-4940-800c-ed95053cff00.

8.19.2. Cortana

Cortana is a voice-activated virtual assistant that debuted with Windows 10. Cortana is a virtual assistant that can respond to questions, explore your computer or the Internet, schedule appointments and reminders, make online purchases, and much more. Cortana is comparable to other voice-activated services like Siri, Alexa, or Google Assistant, with the additional advantage of being able to search your computer's information (Cheng et al., 2016; Đuranec et al., 2019). In Windows 10, click Windows key+S to open Cortana (Figure 8.24).

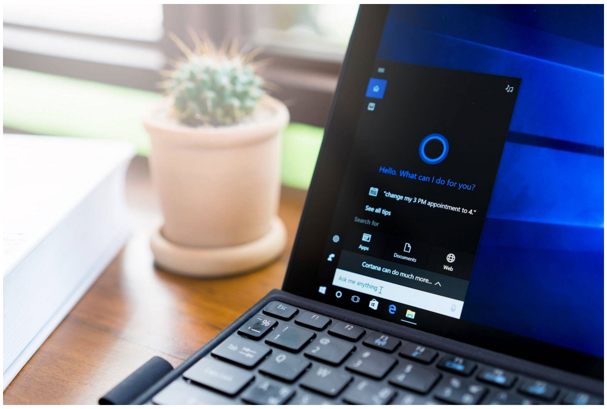

Hello. What can I do for you?
"change my 3 PM appointment to 4."
See all tips
Search for
Apps
Documents
Web
Cortana can do much more...
Ask me anything

Figure 8.24. Cortana interface.

Source: https://cdn.windowsreport.com/wp-content/uploads/2020/06/How-to-block-Cortana-from-starting-in-Windows-10.jpg.

8.19.3. Desktop

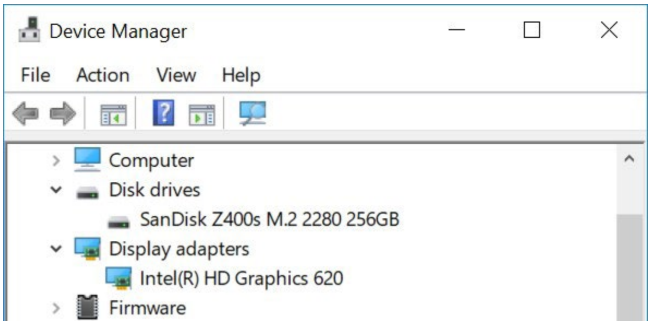
The desktop is a critical component of Windows' standard GUI. It is a container for apps, files, and documents, all of which show as icons. Your desktop is constantly running in the background, alongside any other apps you may be using.

When you turn on your desktop and sign in to Microsoft for the first time, the desktop backdrop, icons, and taskbar are shown. From this point, you may access your computer's installed apps via the Start menu whether by double-clicking any program shortcuts on your desktop.

At any moment, you may access your desktop by hitting Windows key+D to minimize any currently active apps (Lazarescu et al., 2004; Fadhil et al., 2016).

8.19.4. Device Manager

A computer's hardware devices are listed in the Device Manager. The Device Manager allows users to check what hardware is connected, examine, and upgrade hardware drivers, and remove hardware. The Device Manager may be accessed via the Power User Activities Menu (Windows key+X, plus enter M) (Figure 8.25).



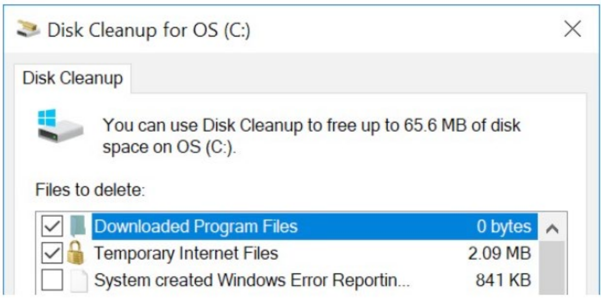
Computer
Disk drives
SanDisk Z400s M.2 2280 256GB
Display adapters
Intel(R) HD Graphics 620
Firmware

Figure 8.25. Device manager interface.

Source: https://www.computerhope.com/issues/ch001967.htm.

8.19.5. Disk Cleanup

The Disk Cleanup tool assists in increasing your computer's available disc space by deleting temporary or unwanted data. Disk Cleanup improves your computer's speed and frees up space on your hard drive for downloads, documents, and programs (Figure 8.26) (Blackman et al., 1989; Bangalee et al., 2012).



You can use Disk Cleanup to free up to 65.6 MB of disk space on OS (C:).

Files to delete:

✓	Downloaded Program Files	0 bytes
✓	Temporary Internet Files	2.09 MB
	System created Windows Error Reportin...	841 KB

Figure 8.26. Disk clean-up display.

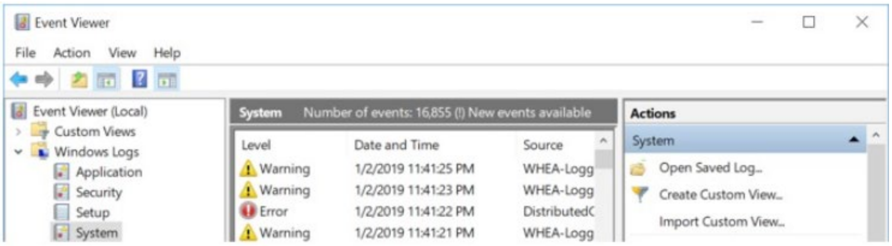
Source: https://www.computerhope.com/issues/ch001967.htm.

Disk Cleanup may be accessed using the File Explorer.Start an Explorer window by pressing Windows key+E:

- Locate this system or My Computer on window's left side and choose it by pressing once;
- Right-click any disc on your desktop on the right side (C, for example);
- Click on properties;
- Select disk cleanup from the general menu.

8.19.6. Event Viewer

The Event Viewer is a system administrator application that shows problems and significant occurrences on your computer. It assists you in troubleshooting sophisticated issues with your Windows PC (Embree et al., 1991; Rajon, 2016). The Power User Tasks Panel (Windows key+X, then V) may be accessed through the Event Viewer (Figure 8.27).



System Number of events: 16,855 (!) New events available

Level	Date and Time	Source
Warning	1/2/2019 11:41:25 PM	WHEA-Logg
Warning	1/2/2019 11:41:23 PM	WHEA-Logg
Error	1/2/2019 11:41:22 PM	DistributedC
Warning	1/2/2019 11:41:21 PM	WHEA-Logg

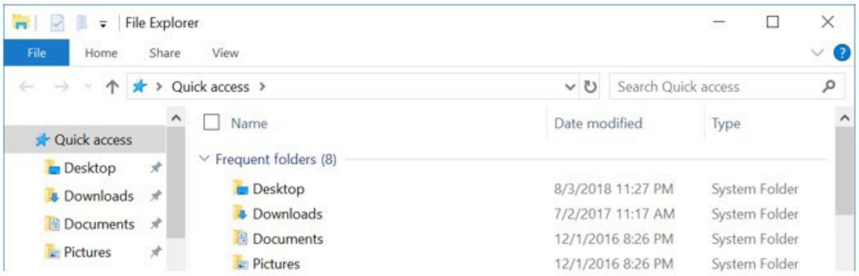
Figure 8.27. Display of the event viewer.

Source: https://www.computerhope.com/issues/ch001967.htm.

8.19.7. File Explorer

The File Explorer, sometimes known as Windows Explorer, gives a graphical representation of the files and directories on your computer. You may view the data of your SSD, hard disks, and removable drives attached to your computer. The File Explorer allows users to find for files and directories and then open, rename, or remove them.

Press Windows key+E to launch a new File Manager window. You may open several Explorer windows concurrently, which is useful for seeing numerous directories at once or copying/moving data between them (Figure 8.28).



Name	Date modified	Type
Desktop	8/3/2018 11:27 PM	System Folder
Downloads	7/2/2017 11:17 AM	System Folder
Documents	12/1/2016 8:26 PM	System Folder
Pictures	12/1/2016 8:26 PM	System Folder

Figure 8.28. File explorer display.

Source: https://www.computerhope.com/issues/ch001967.htm.

8.19.8. Internet Browser

One of the most crucial programs on your computer is your web browser. It may be used to search the Internet for information, read online sites, shop for and purchase items, watch films, play video games, and more. The standard browsing in Windows 10 is Microsoft Edge. Prior window's version from Windows 95 through Windows 8.1, contained Internet Explorer as the default browser. In Windows 10, visit the Menu bar and scroll to the bottom to Microsoft Edge to launch a new Edge web browser.

8.19.9. Microsoft Paint

Microsoft Paint, which has been included with Windows from November 1985, is a straightforward image editor for creating, viewing, and editing digital images. It has basic capability for drawing and painting images, resizing, and rotating photos, and saving images in a variety of file kinds.

To start Microsoft Paint in any version of Windows, hold down the Windows key and enter mspaint (Dehnert and Stepanov, 2000; Mészárosová, 2015). Additionally, it is accessible through the Start menu: in Windows 10, it is featured in Windows Accessories, Paint.

8.19.10. Notepad

Notepad is a straightforward text editor. It allows you to make, examine, and alter text files. For example, you may use Notepad to create a batch file or an HTML web page.

Notepad may be found in the Menu bar under Windows peripherals in Windows 10. Notepad may be launched from the Run box in all windows versions by pressing Windows key+R, typing notepad, then pressing Enter.

8.19.11. Notification Area

The notification area, alternatively referred to as the settings menu, shows the date and time as well as icons for programs that are launched by Windows. Additionally, it displays the status of your Internet access and a loudspeaker icon for volume adjustment (Figure 8.29).

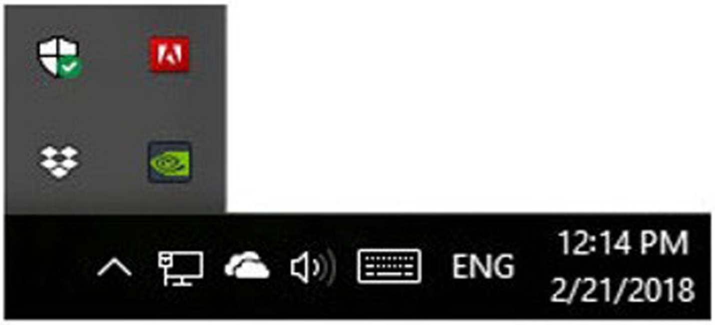


Figure 8.29. Display of notification area.

Source: https://www.computerhope.com/issues/ch001967.htm.

8.19.12. Power User Tasks Menu

The Power User Tasks Menu, which is included in Windows 8 and Windows 10, allows easy access to useful and crucial Windows tools. You may access the Control Panel, Device Manager, File Explorer, Task Manager, and other programs through this menu. Press Windows key+X or right-click the Menu button icon to launch the Power User Tasks Menu.

8.19.13. Registry Editor

The Registry Editor enables you to see and change the Windows system registry. The Registry Editor can be used by computer experts to resolve issues with the Windows pc or installed software (Figure 8.30).

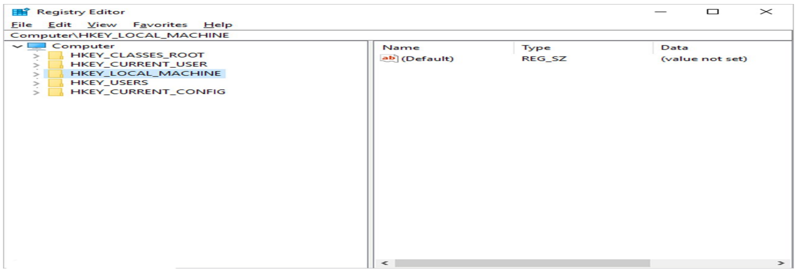


Figure 8.30. Registry editor application.

Source: https://asmwsoft.com/registry-editor-tool.html.

The Registry Editor is available in the Start menu, under Windows Administrative Tools, in Windows 10. Additionally, you may launch it by clicking the Windows key and entering regedit, followed by pressing Enter.

Changing things to the register might result in your programs or system becoming unresponsive. Avoid editing the registry unless you are certain of what you are doing, and always backup your registry before making changes by transferring it to a file (Vogel-Heuser et al., 2014).

8.19.14. Settings

Settings, which is included in Windows 8 and Windows 10, allows you to customize various elements of Windows. You may customize the desktop backdrop, power settings, and external device choices, among other things.

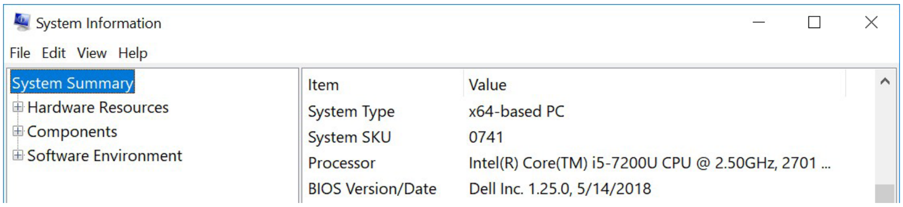
In Windows 10, click Windows key+I to launch Settings. Alternatively, enter the Start menu and select the Gear icon.

8.19.15. Start and Start Menu

The Start menu displays a list among all installed apps and tools on your computer. You may access it by pressing Start just on taskbar's left side. By tapping the Windows key on the keyboard, you may access the Start menu.

8.19.16. System Information

The System Information tool displays information about the computer's hardware and OS. Information regarding your computer's hardware, such as the CPU, RAM, video card, and sound card, can be found. You can also see and change configuration settings, device drivers, applications, and other things (Figure 8.31) (Oliveira et al., 2013).



Item	Value
System Type	x64-based PC
System SKU	0741
Processor	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2701 ...
BIOS Version/Date	Dell Inc. 1.25.0, 5/14/2018

Figure 8.31. System information window.

Source: https://www.computerhope.com/jargon/s/sysinfo.html.

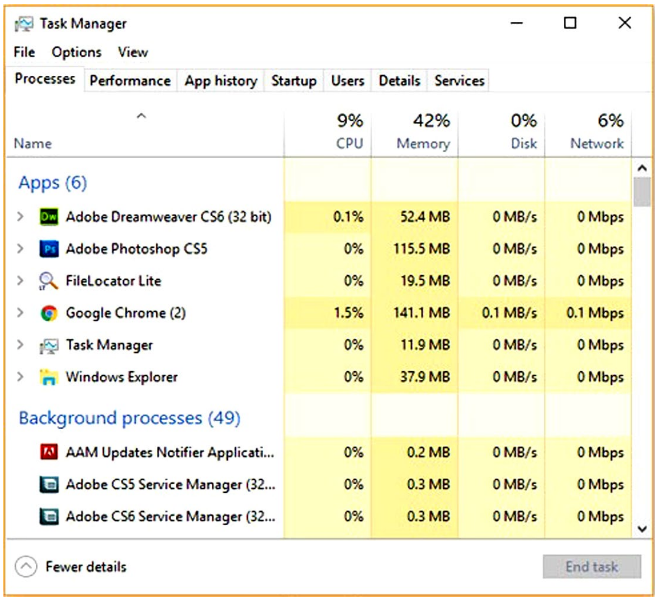
System Information may be found in the Menu bar, in Windows Administrative Tools, in Windows 10. You may also open it by pressing Windows key+R, typing msinfo32, and pressing Enter in the Run box.

8.19.17. Taskbar

The Windows taskbar displays open programs and a Rapid Launch section for quick access to certain apps. The alert area is located to the right of the toolbar and displays the date and time, as well as any background processes.

8.19.18. Task Manager

The Task Manager displays a list of all the programs that are now operating on your computer. You may sort by CPU, RAM, and disc I/O use to discover how many of your system components each program (job) uses. If a program is stuck or not reacting, you may stop the process by right-clicking it in Taskbar and forcing it to close. Ctrl+Shift+Esc will bring up the Task Management at any moment (Figure 8.32).


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
End task

Name	9% CPU	42% Memory	0% Disk	6% Network
Apps (6)				
Adobe Dreamweaver CS6 (32 bit)	0.1%	52.4 MB	0 MB/s	0 Mbps
Adobe Photoshop CS5	0%	115.5 MB	0 MB/s	0 Mbps
FileLocator Lite	0%	19.5 MB	0 MB/s	0 Mbps
Google Chrome (2)	1.5%	141.1 MB	0.1 MB/s	0.1 Mbps
Task Manager	0%	11.9 MB	0 MB/s	0 Mbps
Windows Explorer	0%	37.9 MB	0 MB/s	0 Mbps
Background processes (49)				
AAM Updates Notifier Applicati...	0%	0.2 MB	0 MB/s	0 Mbps
Adobe CS5 Service Manager (32...	0%	0.3 MB	0 MB/s	0 Mbps
Adobe CS6 Service Manager (32...	0%	0.3 MB	0 MB/s	0 Mbps

Figure 8.32. Task manager window.

Source: https://windowsground.com/what-is-task-manager-in-windows-10/.

8.19.19. Windows Search Box

The Windows search box provides an easy method to find documents, images, videos, and programs. Cortana is also integrated into the search bar in Windows 10. The function made its debut in Windows Vista.

By default, the search box is located on your taskbar. If you also don’t see the search box in Windows 10, click right on the toolbar and choose Taskbar settings. Ascertain that the option Use tiny taskbar buttons is disabled. Then, click right the taskbar again and choose Cortana, Show search box from the context menu (Qian and Lau, 2017).

REFERENCES

1. Akbal, E., Günes, F., & Akbal, A., (2016). Digital forensic analyses of web browser records. *J. Softw., 11*(7), 631–637.
2. Archibugi, D., & Pietrobelli, C., (2003). The globalization of technology and its implications for developing countries: Windows of opportunity or further burden? *Technological Forecasting and Social Change, 70*(9), 861–883.
3. Bangalee, M. Z. I., Lin, S. Y., & Miau, J. J., (2012). Wind driven natural ventilation through multiple Windows of a building: A computational approach. *Energy and Buildings, 45*, 317–325.
4. Barney, D., (1994). CC: Mail for Windows 2.0 marred by security flaw. *InfoWorld, 16*(6), 1–2.
5. Bickel, R., Cook, M., Haney, J., Kerr, M., Parker, D. C. T., & Parkes, U. S. N. H., (2002). Guide to securing Microsoft Windows XP. *National Security Agency,* 1–129.
6. Bird, C., Nagappan, N., Devanbu, P., Gall, H., & Murphy, B., (2009). Does distributed development affect software quality? an empirical case study of Windows vista. In: *2009 IEEE 31st International Conference on Software Engineering* (pp. 518–528). IEEE.
7. Blackman, C. F., Kinney, L. S., House, D. E., & Joines, W. T., (1989). Multiple power-density Windows and their possible origin. *Bioelectromagnetics: Journal of the Bioelectromagnetics Society, The Society for Physical Regulation in Biology and Medicine, The European Bioelectromagnetics Association, 10*(2), 115–128.
8. Bolosky, W. J., Corbin, S., Goebel, D., & Douceur, J. R., (2000). Single instance storage in Windows 2000. In: *Proceedings of the 4th USENIX Windows Systems Symposium* (pp. 13–24).
9. Campbell, G., (1991). Word for Windows 2.0: Getting better all the time. *PC World, 9*(12), 91–92.
10. Campbell, G., (1992). Windows word processors. *PC World, 10*(4), 146–161.
11. Chau, P. Y., & Hui, K. L., (1998). Identifying early adopters of new IT products: A case of Windows 95. *Information & Management, 33*(5), 225–230.
12. Cheng, Y., Jiang, C., & Shi, J., (2016). A fall detection system based on SensorTag and Windows 10 IoT core. In: *2015 International*

Conference on Mechanical Science and Engineering (pp. 238–244). Atlantis Press.

13. Chien, C. F., Lin, K. Y., & Yu, A. P. I., (2014). User-experience of tablet operating system: An experimental investigation of Windows 8, iOS 6, and, android 4.2. *Computers & Industrial Engineering, 73*, 75–84.
14. Corregedor, M., & Von, S. S., (2013). Windows 8; 32 bit—Improved security? In: *2013 Africon* (pp. 1–5). IEEE.
15. Cota-Robles, E., & Held, J. P., (1999). A comparison of Windows driver model latency performance on Windows NT and Windows 98. In: *OSDI* (pp. 159–172).
16. Dehnert, J. C., & Stepanov, A., (2000). Fundamentals of generic programming. In: *Generic Programming* (pp. 1–11). Springer, Berlin, Heidelberg.
17. Dong, C., (1999). PowderX: Windows-95-based program for powder x-ray diffraction data processing. *Journal of Applied Crystallography, 32*(4), 830–838.
18. Đuranec, A., Topolčić, D., Hausknecht, K., & Delija, D., (2019). Investigating file use and knowledge with Windows 10 artifacts. In: *2019 42nd International Convention on Information and Communication Technology, Electronics, and Microelectronics (MIPRO)* (pp. 1213–1218). IEEE.
19. Embree, P. M., Kimble, B., & Bartram, J. F., (1991). *C Language Algorithms for Digital Signal Processing,* 15–20.
20. Eterovic-Soric, B., Choo, K. K. R., Mubarak, S., & Ashman, H., (2017). Windows 7 antiforensics: A review and a novel approach. *Journal of Forensic Sciences, 62*(4), 1054–1070.
21. Fadhil, M. S., Alkawaz, M. H., Rehman, A., & Saba, T., (2016). Writers identification based on multiple Windows features mining. *3D Research, 7*(1), 1–6.
22. Ganapathi, A., Ganapathi, V., & Patterson, D. A., (2006). Windows XP kernel crash analysis. In: *LISA* (Vol. 6, pp. 49–159).
23. Gao, H., Jia, W., Liu, N., & Li, K., (2013). The hot-spots problem in Windows 8 graphical password scheme. In: *International Symposium on Cyberspace Safety and Security* (pp. 349–362). Springer, Cham.
24. Gratze, G., Fortin, J., Holler, A., Grasenick, K., Pfurtscheller, G., Wach, P., & Skrabal, F., (1998). A software package for non-invasive, real-time beat-to-beat monitoring of stroke volume, blood pressure,

total peripheral resistance and for assessment of autonomic function an updated and improved software version for Windows 95/NT and the complete biosignal electronics (ECG, ICG, beat-to-beat, and oscillometric blood pressure and pulse oxymetry) will be supplied in a compact instrument by: CNSystems Medical Equipment Inc. Heinrichstrasse 22 A-8010 Graz, Austria, Europe. Tel.: +43/316/3631-0; Fax:+ 43/316. *Computers in Biology and Medicine, 28*(2), 121–142.

25. Guo, P. J., Zimmermann, T., Nagappan, N., & Murphy, B., (2010). Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (Vol. 1, pp. 495–504).
26. Hamadani, J. D., Tofail, F., Nermell, B., Gardner, R., Shiraji, S., Bottai, M., & Vahter, M., (2011). Critical Windows of exposure for arsenic-associated impairment of cognitive function in pre-school girls and boys: A population-based cohort study. *International Journal of Epidemiology, 40*(6), 1593–1604.
27. Hargreaves, C., Chivers, H., & Titheridge, D., (2008). Windows vista and digital investigations. *Digital Investigation, 5*(1, 2), 34–48.
28. Harris, D. C., (1999). *Materials for Infrared Windows and Domes: Properties and Performance* (Vol. 158, pp. 25–40). SPIE press.
29. Ho, W. K., Ang, J. C., & Lim, A., (2001). A hybrid search algorithm for the vehicle routing problem with time Windows. *International Journal on Artificial Intelligence Tools, 10*(3), 431–449.
30. Hofmeester, K., & Wolfe, J., (2012). Self-revealing gestures: Teaching new touch interactions in Windows 8. In: *CHI'12 Extended Abstracts on Human Factors in Computing Systems* (pp. 815–828).
31. Huxford, D. C., (1993). Windows for workgroups. *Journal of Financial Planning, 6*(2), 52.
32. Inglot, B., & Liu, L., (2014). Enhanced timeline analysis for digital forensic investigations. *Information Security Journal: A Global Perspective, 23*(1, 2), 32–44.
33. Khatri, Y., (2015). Forensic implications of system resource usage monitor (SRUM) data in Windows 8. *Digital Investigation, 12*, 53–65.
34. Lallie, H. S., & Briggs, P. J., (2011). Windows 7 registry forensic evidence created by three popular BitTorrent clients. *Digital Investigation, 7*(3, 4), 127–134.

35. Laric, M. V., (1995). Day-Timer organizer 1.0 for Windows. *Journal of Consumer Marketing, 12*(3), 67–70.

36. Lazarescu, M. M., Venkatesh, S., & Bui, H. H., (2004). Using multiple Windows to track concept drift. *Intelligent Data Analysis, 8*(1), 29–59.

37. Lee, T., Mitschke, K., Schill, M. E., & Tanasovski, T., (2011). *Windows PowerShell 2.0 Bible* (Vol. 725, pp. 457–472). John Wiley & Sons.

38. Li, P. L., Ni, M., Xue, S., Mullally, J. P., Garzia, M., & Khambatti, M., (2008). Reliability assessment of mass-market software: Insights from Windows vista®. In: *2008 19th International Symposium on Software Reliability Engineering (ISSRE)* (pp. 265–270). IEEE.

39. Lin, S. W., & Vincent, F. Y., (2012). A simulated annealing heuristic for the team orienteering problem with time Windows. *European Journal of Operational Research, 217*(1), 94–107.

40. Lio, P. A., & Nghiem, P., (2004). Interactive atlas of dermoscopy: Giuseppe argenziano, MD, H. Peter Soyer, MD, Vincenzo De Giorgio, MD, Domenico Piccolo, MD, Paolo Carli, MD, Mario Delfino, MD, Angela Ferrari, MD, Rainer Hofmann-Wellenhof, MD, Daniela Massi, MD, Giampiero Mazzocchetti, MD, Massimiliano Scalvenzi, MD, and Ingrid H. Wolf, MD, Milan, Italy, 2000, Edra Medical Publishing and New Media. $290.00. ISBN 88-86457-30-8. CD-ROM requirements (minimum): Pentium 133 MHz, 32-Mb RAM, 24X CD-ROM drive, 800× 600 resolution. *Journal of the American Academy of Dermatology, 50*(5), 208, 807, 808.

41. Ma, X., Buffler, P. A., Gunier, R. B., Dahl, G., Smith, M. T., Reinier, K., & Reynolds, P., (2002). Critical Windows of exposure to household pesticides and risk of childhood leukemia. *Environmental Health Perspectives, 110*(9), 955–960.

42. Mehreen, S., & Aslam, B., (2015). Windows 8 cloud storage analysis: Dropbox forensics. In: *2015 12th International Bhurban Conference on Applied Sciences and Technology (IBCAST)* (pp. 312–317). IEEE.

43. Mészárosová, E., (2015). Is python an appropriate programming language for teaching programming in secondary schools. *International Journal of Information and Communication Technologies in Education, 4*(2), 5–14.

44. Naik, D. C., (2004). Inside Windows storage: Server storage technologies for Windows 2000, Windows server 2003 and beyond. *Computing Reviews, 45*(8), 468–469.

45. Narayan, S., Feng, T., Xu, X., & Ardham, S., (2009). Network performance evaluation of wireless IEEE802. 11n encryption methods on Windows vista and Windows server 2008 operating systems. In: *2009 IFIP International Conference on Wireless and Optical Communications Networks* (pp. 1–5). IEEE.
46. Narayan, S., Shang, P., & Fan, N., (2009). Performance evaluation of ipv4 and ipv6 on Windows vista and Linux ubuntu. In: *2009 International Conference on Networks Security, Wireless Communications and Trusted Computing* (Vol. 1, pp. 653–656). IEEE.
47. Nolze, G., & Kraus, W., (1998). PowderCell 2.0 for Windows. *Powder Diffraction, 13*(4), 256–259.
48. Odell, D., & Chandrasekaran, V., (2012). Enabling comfortable thumb interaction in tablet computers: A Windows 8 case study. In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* (Vol. 56, No. 1, pp. 1907–1911). Sage CA: Los Angeles, CA: SAGE Publications.
49. Oliveira, O. L., Monteiro, A. M., & Roman, N. T., (2013). Can natural language be utilized in the learning of programming fundamentals? In: *2013 IEEE Frontiers in Education Conference (FIE)* (pp. 1851–1856). IEEE.
50. Pfeiffer, P., Scott, S. L., & Shukla, H., (2003). ORNL-RSH package and Windows '03 PVM 3.4. In: *European Parallel Virtual Machine/ Message Passing Interface Users' Group Meeting* (pp. 388–394). Springer, Berlin, Heidelberg.
51. Prentice, A. M., Ward, K. A., Goldberg, G. R., Jarjou, L. M., Moore, S. E., Fulford, A. J., & Prentice, A., (2013). Critical Windows for nutritional interventions against stunting. *The American of Clinical Nutrition, 97*(5), 911–918.
52. Purcell, D. M., & Lang, S. D., (2008). Forensic artifacts of Microsoft Windows vista system. In: *International Conference on Intelligence and Security Informatics* (pp. 304–319). Springer, Berlin, Heidelberg.
53. Qian, C., & Lau, K. K., (2017). Enumerative variability in software product families. In: *2017 International Conference on Computational Science and Computational Intelligence (CSCI)* (pp. 957–962). IEEE.
54. Rajesh, B., Reddy, Y. J., & Reddy, B. D. K., (2015). A survey paper on malicious computer worms. *International Journal of Advanced Research in Computer Science and Technology, 3*(2), 161–167.

55. Rajon, S. A., (2016). *Fundamentals of Computer Programming with C* (Vol. 10, pp. 25–35). SA AHSAN RAJON.

56. Ray, E., & Schultz, E. E., (2007). An early look at Windows vista security. *Computer Fraud & Security, 2007*(1), 4–7.

57. Rushinek, A., & Rushinek, S. F., (1997). Project management software feature profitability: Windows, networks, mainframes, filtered task diagrams, schedules, and calendars. *Journal of Computer Information Systems, 37*(4), 48–55.

58. Scoville, R., (1993). 1-2-3 for Windows 2.0: Don't count Lotus out. *PC World, 11*(1), 149–149.

59. Sechrest, S., & Fortin, M., (2001). Windows XP Performance. *Microsoft, dated Jun, 1*, 20.

60. Singh, B., & Singh, U., (2016). A forensic insight into Windows 10 jump lists. *Digital Investigation, 17*, 1–13.

61. Singh, B., & Singh, U., (2017). A forensic insight into Windows 10 Cortana search. *Computers & Security, 66*, 142–154.

62. Smith, W. R., (1996). HSC chemistry for Windows, 2.0. *Journal of Chemical Information and Computer Sciences, 36*(1), 151, 152.

63. Stiegler, M., Karp, A. H., Yee, K. P., Close, T., & Miller, M. S., (2006). Polaris: Virus-safe computing for Windows XP. *Communications of the ACM, 49*(9), 83–88.

64. Sullivan, K., (1996). The Windows 95 user interface: A case study in usability engineering. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 473–480).

65. Swift, M. M., Hopkins, A., Brundrett, P., Van, D. C., Garg, P., Chan, S., & Jensenworth, G., (2002). Improving the granularity of access control for Windows 2000. *ACM Transactions on Information and System Security (TISSEC), 5*(4), 398–437.

66. Talebi, J., Dehghantanha, A., & Mahmoud, R., (2012). Introducing and analysis of the Windows 8 event log for forensic purposes. In: *Computational Forensics* (pp. 145–162). Springer, Cham.

67. Thomas, S., Sherly, K. K., & Dija, S., (2013). Extraction of memory forensic artifacts from Windows 7 ram image. In: *2013 IEEE Conference on Information & Communication Technologies* (pp. 937–942). IEEE.

68. Uslan, M. M., & Su, J. C., (1997). A review of two screen magnification programs for Windows 95: Magnum 95 and LP-Windows. *Journal of Visual Impairment & Blindness, 91*(5), 9–13.

69. Uzunboylu, H., Bicen, H., & Cavus, N., (2011). The efficient virtual learning environment: A case study of web 2.0 tools and Windows live spaces. *Computers & Education, 56*(3), 720–726.

70. Venčkauskas, A., Damaševičius, R., Jusas, N., Jusas, V., Maciulevičius, S., Marcinkevičius, R., & Toldinas, J., (2015). Investigation of artifacts left by BitTorrent client in Windows 8 registry. *Science and Education, 3*(2), 25–31.

71. Vogel-Heuser, B., Rehberger, S., Frank, T., & Aicher, T., (2014). Quality despite quantity—Teaching large heterogeneous classes in C programming and fundamentals in computer science. In: *2014 IEEE Global Engineering Education Conference (EDUCON)* (pp. 367–372). IEEE.

72. Warner, P. D., (2001). Windows ME. *The CPA Journal, 71*(1), 64.

73. Westerlund, A., & Danielsson, J., (2001). Heimdal and Windows 2000 Kerberos-how to get them to play together. In: *USENIX Annual Technical Conference, FREENIX Track* (pp. 267–272).

74. Whitehouse, O., (2007). An analysis of address space layout randomization on Windows vista. *Symantec Advanced Threat Research,* 1–14.

75. Zhang, S., Wang, L., Zhang, R., & Guo, Q., (2010). Exploratory study on memory analysis of Windows 7 operating system. In: *2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)* (Vol. 6, pp. V6–373). IEEE.

76. Zhang, W., Lu, L., Peng, J., & Song, A., (2016). Comparison of the overall energy performance of semi-transparent photovoltaic Windows and common energy-efficient Windows in Hong Kong. *Energy and Buildings, 128*, 511–518.

77. Zimmermann, T., Nagappan, N., & Williams, L., (2010). Searching for a needle in a haystack: Predicting security vulnerabilities for Windows vista. In: *2010 Third International Conference on Software Testing, Verification, and Validation* (pp. 421–428). IEEE.

INDEX

Key Dynamics in Computer Programming

A program is created by first defining a task and then expressing it in a computer language that is appropriate for the application. The specification is then converted into a coded program that can be directly executed by the machine on which the task is to be performed, usually in numerous steps. Machine language refers to the coded program, whereas problem-oriented languages refer to languages that are ideal for original formulation. C, Python, and C++ are only a few of the many problem-solving languages that have been invented. Computers come with a variety of programs that are meant to help users do jobs and improve system performance. The operating system (OS), which is a collection of programs, is as crucial to the operation of a computer system as its hardware. Current technology allows some operating features to be built into a computer's central processing unit as fixed programs (introduced by client orders) at the time of production. The operating system may have control over user programs during execution, such as when a time-sharing monitor suspends one program and activates another, or when a user program is begun or terminated, such as when a scheduling software chooses which user program will be executed next. Certain operating-system programs, on the other hand, run as stand-alone modules to make the programming process easier. While translators (assemblers or compilers) convert an entire program from one language to another, interpreters execute a program sequentially. Interpreters translate at each step and debuggers execute a program piecemeal and monitor various circumstances, allowing the program to check whether the program's operation is correct or not.
This book aims to help the student understand computer programming by presenting the fundamentals of computer hardware and software, computer programs, operating systems, major programming languages, and an introduction to Windows operating systems. Chapter 1 introduces the readers to the fundamentals of computers and computer programs. Chapter 2 deals with the classification of computer programs. Chapter 3 discusses the fundamentals of programming languages. Chapters 4 and 5 introduce the readers to two major languages: Python and C language. Chapter 6 illustrates the idea of dynamic programming and its uses. Chapter 7 focuses on the fundamentals of different operating systems. Finally, Chapter 8 deals with the timeline of Windows with a focus on its features. We have not hesitated to be prescriptive: to claim that accumulated experience shows that certain constructs are to be preferred, and others to be avoided or at least used with caution Of course, any book on programming languages should not be taken as a reference manual for any particular language. The book equips you with insights so that you can learn to analyze languages and not to study the peculiarities of any language in depth. Nor is the book a guide to the choice of a language for any particular project. The goal is to supply the student with the conceptual tools needed to make such a decision.

Adele Kuzmiakova is a machine learning engineer focusing on solving problems in machine learning, deep learning, and computer vision. Adele currently works as a senior machine learning engineer at Ifolor focusing on creating engaging photo stories and products. Adele attended Cornell University in New York, United States for her undergraduate studies. She studied engineering with a focus on applied math. Some of the deep learning problems Adele worked on include predicting air quality from public webcams, developing a real-time human movement tracking, and using 3D computer vision to create 3D avatars from selfies in order to bring online clothes shopping closer to reality. She is also passionate about exchanging ideas and inspiring other people and acted as a workshop organizer at Women in Data Science conference in Geneva, Switzerland.

ISBN 978-1-77469-638-5

AP | ARCLER PRESS