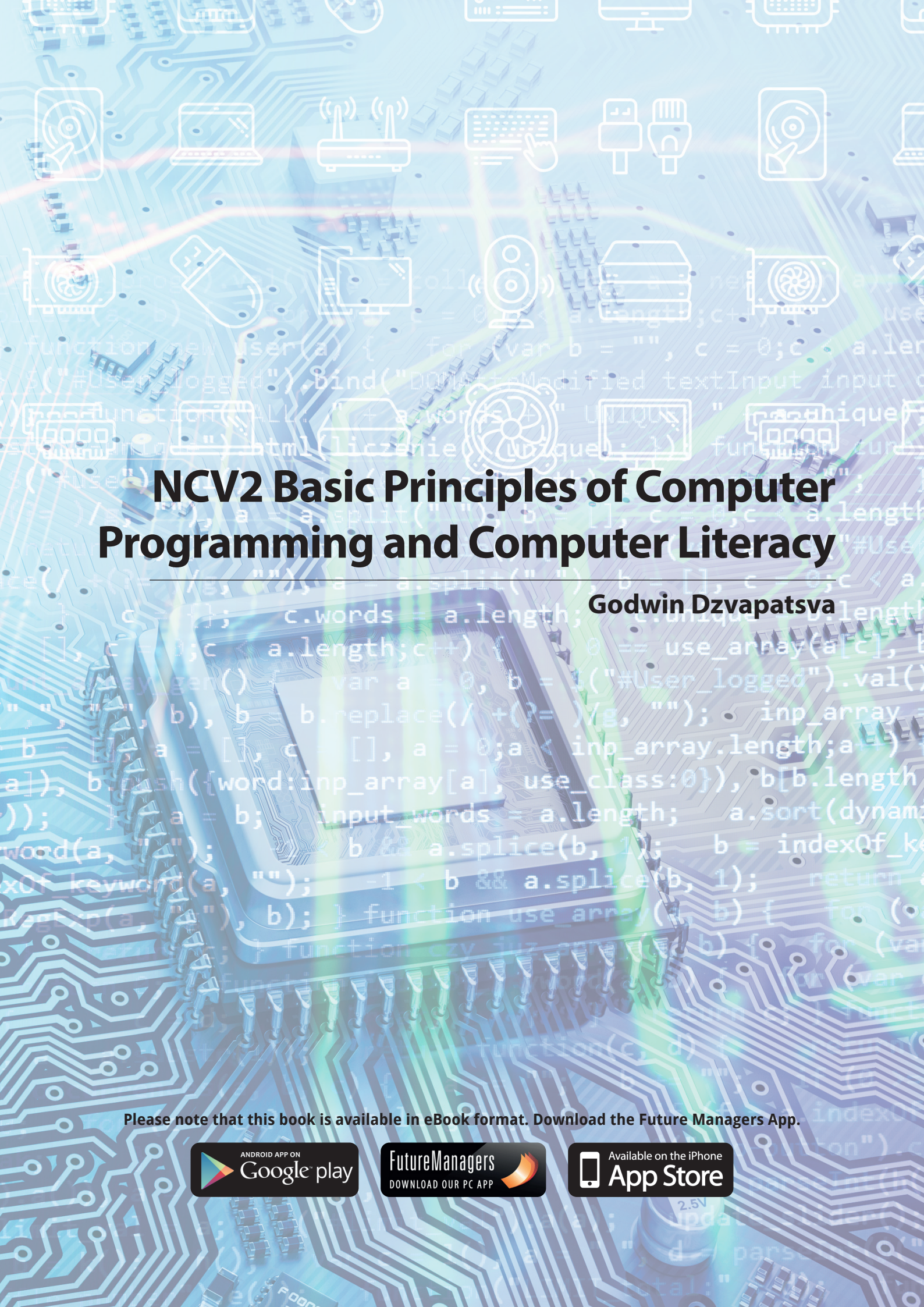


NCV2 Basic Principles of Computer Programming and Computer Literacy

Godwin Dzvapatsva

ISBN 978-0-63911-092-9

First edition 2022

FutureManagers
SIYAFUNDA • SIYAKHULA

Published by
Future Managers (Pty) Ltd
PO Box 13194, Mowbray, 7705
Tel (021) 462 3572
Fax (021) 462 3681
E-mail: info@futuremanagers.com
Website: www.futuremanagers.com

Contents

Icon key

Whenever you see the following ICONS in this text book, this is what they mean:

INDIVIDUAL ACTIVITY

Indicates an activity to be done alone.

PAIR ACTIVITY

Indicates an activity to be done in pairs.

GROUP ACTIVITY

Indicates an activity to be done in groups.

INVESTIGATION

Indicates an activity that requires investigation.

TASK

Practise this skill in the simulated environment or practicum room.

DID YOU KNOW?

Gives information that you might not have been aware of.

EXAMPLE

Highlights examples that have been given in the text.

eLINK

Shows a hyperlink to a related website.

NOTE

Explains in more detail.

PAUSE FOR THOUGHT

Gives you information to think about.

REMEMBER

Highlights important information.

SUMMATIVE ASSESSMENT

This is the activity that you will need to complete and tear out to be handed in to your lecturer as part of your assessment mark.

TIP

Gives useful information.

VOCABULARY

This is an explanation of difficult or typical words.

Module 1

Computer hardware and software

After you have completed this module, you should be able to:

- describe the term *hardware*;
- describe how hardware communicate;
- classify different hardware components in relation to the evolution of technology;
- list different hardware components with relation to their use;
- define the term *input hardware* and provide examples;
- describe the purpose of input hardware;
- describe the purpose of output hardware;
- describe the purpose of processing hardware;
- describe the purpose of storage hardware;
- define the term *processing hardware* and provide examples;
- define the term *output hardware* and provide examples;
- describe the typical components of the system unit;
- discuss the concept *modular design of a computer-based system*;
- define the term *single-board computer*;
- list examples of single-board computers;
- discuss the purpose and uses of single-board computers;
- present a typical breakdown of the components of a single-board computer;
- define the term *shell*;
- launch a new Command prompt in Windows;
- use the `help` command to list some common commands;
- expand a Windows file path and explain each element;
- list the contents of the current folder using the `dir` command;
- change directly location using the `cd` command;
- create a new folder using the `mkdir` command;
- remove a folder using the `rmdir` command;
- remove a file using the `del` command;
- rename a file using the `rename` command and copy a file using the copy command;
- clear the Command prompt screen using the `cls` command;
- run an executable file from the command line;
- list the system information using the `systeminfo` command; and
- copy the path of any file by dropping the file in the Command prompt.

Introduction

The use of computers in schools, banks, shops, railway stations, hospitals, and your own home helps make our work easier and faster. It is very likely that you have used, seen or read about computers before. Because they play such an important role in most of our lives, it is important to understand how computers function.

1.1 Hardware

VOCABULARY

Software – *set of instructions, data or programs used to operate computers and execute specific tasks*

Computer – *electronic device that accepts data as an input and processes it according to instructions to produce an output; computers are made up of hardware and software*

In this section, we will define the term *hardware* and explain how hardware interacts with **software** to accomplish given work.

Hardware refers to the physical components of a **computer** or a machine that we can see and touch. It contains circuit board, integrated circuits (ICs), or other electronics of a computer system. Figure 1.1 shows the different hardware components.

Figure 1.1: Computer hardware

1.1.1 Hardware communication

Information is transmitted from one point to another through signals. There are two main types of signals used in electronics: analogue and digital signals.

An analogue signal is a continuous signal and can have infinite values in a given period. When plotted on a voltage vs. time graph, an analogue signal should produce a smooth and continuous curve. Digital signals are used in all digital electronics, including computing equipment and data transmission devices. When plotted on a voltage vs. time graph, digital signals are one of two values (0s or 1s).

Figure 1.2 illustrates analogue and digital signals plotted on a graph.

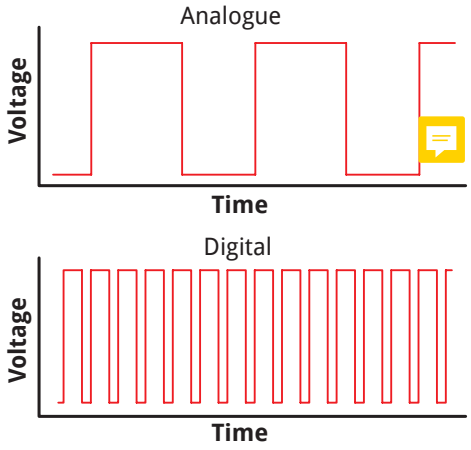


Figure 1.2: Analogue and digital signals

So why are digital signals better than analogue signals?

Digital signals have a better transmission rate, less impact of noise and less distortion. They are less expensive and more flexible.

Character sets

The binary number system is widely used in computing due to its simplicity. Computers do not understand language and numbers in the same way people do. Switches and electrical signals that are either on or off are all they have to deal with. By using switches that can only be turned on or off, binary systems are a natural choice for encoding instructions or storing values. Binary code represents 'Off' as 0 and 'On' as 1. For counting, it is more efficient to represent the number of micro transistors in terms of 1s and 0s. A single transistor is known as a bit, which stands for *binary digit*. A byte is 8 bits in a row representing any number between 0 and 255. Computer hardware converts high-level language (your input) into a low-level language (binary) and is then capable of grouping the binary digits to display what is readable.

Let us look at the following representation below.

0	1	1	0	0	1
32	16	8	4	2	1
2^5	2^4	2^3	2^2	2^1	2^0

To get the answer:
We multiply the values by the binary and add to the next digit as follows:

$(32 \times 0) + (16 \times 1) + (8 \times 1) + (4 \times 0) + (2 \times 0) + (1 \times 1)$

The above calculation will be reduced to the following:

$16 + 8 + 1 = 25_{10}$

So, 011001_2 is equivalent to 25_{10}.

With binary, the largest number we can represent values from 0 to 255 using eight digits. All the 1s represent an ON state and all the 0s represent an OFF state. Hardware responds to electrical signals when a switch is turned on or off. One switch is like one bit, so a bit symbolises the smallest amount of information that can be configured. A byte is made up of eight switches, or eight bits. Instructions are made up of strings of these bits, which may be read by the appropriate hardware. Look at the steps followed to change decimal numbers into binary numbers:

CONVERT DECIMAL TO BINARY

1. Divide the number by 2.
2. Get the integer quotient for the next iteration.
3. Get the remainder for the binary digit.
4. Repeat the steps until the quotient is equal to 0.

EXAMPLE 1.1

Convert 13_{10} to binary.

Solution

DIVISION BY 2	QUOTIENT	REMAINDER	BIT NUMBER
13/2	6	1	0
6/2	3	0	1
3/2	1	1	2
$\frac{1}{2}$	0	1	3

$\therefore 13_{10} = 110_{12}$

EXAMPLE 1.2

Convert 174_{10} to binary.

Solution

DIVISION BY 2	QUOTIENT	REMAINDER	BIT NUMBER
174/2	87	0	0
87/2	43	1	1
43/2	21	1	2
21/2	10	1	3
10/2	5	0	4
5/2	2	1	5
2/2	1	0	6
$\frac{1}{2}$	0	1	7

$\therefore 174_{10} = 10101110_{2}$

CONVERT BINARY TO DECIMAL

For binary number with n digits:

$d_{n-1} \ldots d_3\, d_2\, d_1\, d_0$

The decimal number is equal to the sum of binary digits (d_n) times their power of 2 (2^n):

Decimal = $d_0 \times 2^0 + d_1 \times 2^1 + d_2 \times 2^2 + \ldots$

EXAMPLE 1.3

Find the decimal value of 111001_2.

Solution

BINARY	1	1	1	0	0	1
POWER OF 2	2^5	2^4	2^3	2^2	2^1	2^0

$\therefore 111001_2 = 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

$= 57_{10}$

A code is created when you touch a key on your computer's keyboard. Every character on the keyboard has a unique binary code. Computing systems are able to communicate meaningfully because they use similar codes (more or less).

Besides binary code, the other character representation forms are as follows:

- ASCII-codes – All possible key presses on a computer keyboard are represented by the American Standard Code for Information Interchange, or ASCII (pronounced *ass-key*). It is the set of codes used by personal computers.
- EBCDIC – The Extended Binary Coded Decimal Interchange Code is a character set used by older 'mainframes'.
- Unicode – Unicode uses 16 bits to give about 65 000 unique codes that have been allocated to symbols.

eLINK

Visit this link to learn more about analogue and digital signals:

~~https://youtu.be/ZWdT-6Ld71Q~~

Activity 1.1 **INDIVIDUAL ACTIVITY**

1. Convert the following decimal numbers to their binary equivalents:

1.1 96

1.2 3

1.3 164 (2 × 3) (6)

Activity 1.1 (continued)

2. Why are digital signals better than analogue signals? (2)
3. Which of the following characterises an analogue quantity?
   - **A** Discrete levels represent changes in a quantity.
   - **B** Its values follow a logarithmic response curve.
   - **C** It can be described with a finite number of steps.
   - **D** It has a continuous set of values over a given range. (1)

TOTAL: 9

1.1.2 Hardware components in relation to the evolution of technology

Many of the components in today's personal computers are the same as they were in the 1990s. However, a lot has happened since then. This shift is visible in the way the components interact and the speed with which activities are done.

With each generation of computers, the technology that a computer is/was based on has progressed. Originally, the term *generation* related to physical technologies, but it now encompasses both hardware and software. A device can be updated in a variety of ways, including making it smaller, less expensive, smarter or more powerful.

There are five generations of computers, which are listed in Table 1.1 with the approximate period:

GENERATION	DESCRIPTION
First generation (1946–1959)	Based on valves (vacuum tubes), e.g. UNIVAC
Second generation (1960–1965)	Consisted of transistors, e.g. IBM 1620
Third generation (1966–1971)	Used ICs, e.g. IBM 360
Fourth generation (1972–1980)	Very large-scale ICs, e.g. DEC10, STAR 1000
Fifth generation (1981–present)	Ultra-large-scale integration, artificial intelligence (AI), e.g. desktop, laptops

Table 1.1: Computer generations

In general, computer components still function the same way they used to. The motherboard remains the computer's hub, with all connections pointing to it; the processor processes instructions, random-access memory (RAM) stores data for quick access, and hard drives still store data long term.

Moore's law is mentioned when discussing computer evolution. It basically states that:

- the number of ICs in microprocessors will double within every two years; and
- the more ICs, or transistors, a chip has, the faster it is going to be.

Some key changes in computer evolution witnessed includes the following:

- Increase in storage capacity
- The rise of the solid-state drive
- The miniaturisation of hardware components.

Increase in storage capacity

Hard drive disk storage has increased dramatically since the 1990s. We now measure drives in terabytes instead of megabytes. The latest interfaces for transmitting data have also made a big difference. Parallel ATA systems are capable of 133 MB per second, while Serial ATA, or SATA, is currently capable of 6 GB per second (768 MB).

The rise of the solid-state drive

In recent years, computers have used solid-state or flash memory technology to store data instead of hard drives, enabling them to access data even faster.

Figure 1.3: A solid-state drive

The miniaturisation of hardware components

With the rise of smartphones, computer components have become smaller. But even in smartphones, many of the same components work the same way as they would do in full-sized computers.

Classification of computers

According to usage and functionalities, computers can be classified as analogue, digital or hybrid computers. In this book, we will concentrate on various classes of digital computers. There are four types of digital computers: supercomputers, mainframes, minicomputers and microcomputers.

- Supercomputers – Large and require a lot of space for installation. Because they are faster and more expensive, they are mainly used for calculations that require advanced processing.
- Mainframe computers – These are smaller than supercomputers, but still relatively large. They also perform complex calculations. Mainframe computers are commonly found in banks and educational sectors.
- Microcomputers – Inexpensive and support multi-user platforms. These computers are used by small organisations. Microcomputers are called *personal computers*.
- Minicomputers – Cheaper and easy to carry. Notebooks and tablets are examples of minicomputers.

Computers continue to evolve due to technological innovation. Future technologies will have an enormous impact on society. The Fourth Industrial Revolution (4IR) has been triggered by these revolutionary technologies and is causing a major transformation in almost every aspect of our lives and in the way business is conducted. The 4IR brings together advances in AI, robotics, the Internet of Things, blockchain, virtual reality, big data analytics, high-speed mobile Internet, genetic engineering, quantum computing, and more.

~~Figure 1.4~~ gives a summary of the key components of the 4IR.

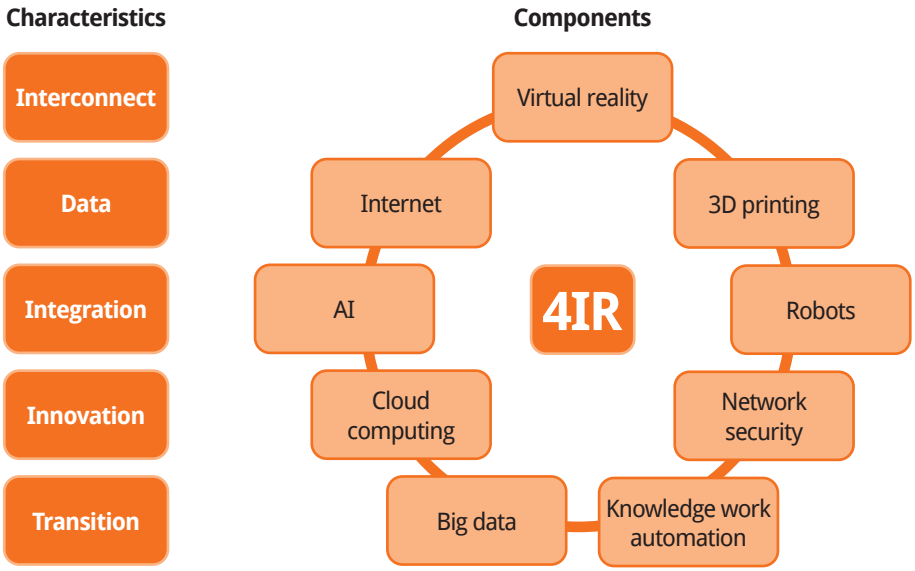


Figure 1.4: Components of 4IR

1.1.3 Hardware components and their uses

Hardware devices can be classified into five distinct categories:

- Input devices: for raw data input
- Processing devices: to process raw data instructions into information
- Output devices: to disseminate data and information
- Storage devices: for data and information retention
- Communication devices: for point-to-point communication.

Figure 1.5 provides a quick glance of hardware.

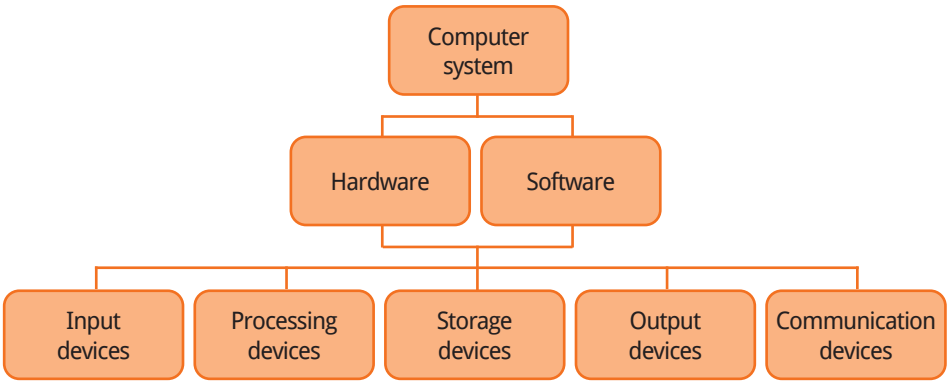


Figure 1.5: Quick glance of hardware

Activity 1.2 **INDIVIDUAL ACTIVITY**

1. Define the term *computer*. (2)
2. What are the TWO constituencies that make up a computer? (2)
3. Describe the term *hardware* in relation to computer systems. (2)
4. What is the difference between an *analogue signal* and a *digital signal*? (4)
5. List FOUR classes of computers. (4)
6. State whether the following statement is true or false: A byte consists of eight bits. (1)
7. List the FIVE hardware components and explain each of them. (5)

TOTAL: 20

1.2 Hardware components of a typical system

This section explains the purposes of input hardware, processing hardware, storage hardware and output hardware. Examples are presented for each hardware component. Hardware components can be classified as internal or external. A motherboard, a power supply and a central processing unit (CPU) are examples of internal components found within a computer. External components, often known as peripherals or peripheral devices, are those that connect to the outside of the computer. Peripherals include input devices, output devices and storage hardware.

1.2.1 Input hardware

Input hardware or input devices are devices used to enter data or instructions into the CPU. Input devices are classified according to how they enter data. They input data such as text, images and audio-visual recordings. In addition, they help transfer files between computers. The keyboard is probably the most commonly used input device.

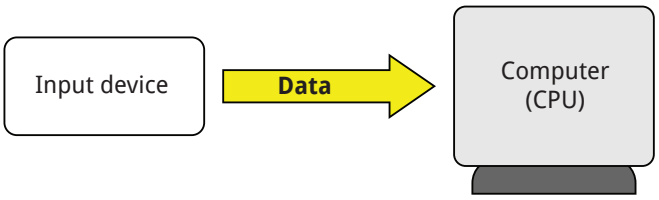


Figure 1.6: Relationship between input device and the computer

Table 1.2 shows the different categories of input devices.

INPUT TYPE	EXAMPLES
Pointing device	Mouse, touchpad, touchscreen, multi-touchscreen, pen input, motion sensor, graphics tablet, interactive smartboard, fingerprint scanner
Game controller	Joystick, gamepad, steering wheel
Audio input device	Microphone, midi keyboard
Bluetooth peripheral	Keyboard, mouse, headset, gamepad, printer

INPUT TYPE	EXAMPLES
Visual and imaging device	Webcam, digital camera, digital camcorder, TV capture card, biometric scanner, barcode reader
Network device	Ethernet hardware, bluetooth/wireless hardware

Table 1.2: Categories and examples of input hardware

1.2.2 Purpose of input hardware

The purposes of input hardware are the following:

- Input devices translate user-friendly inputs into machine-understandable inputs.
- An input device helps to enter data or instructions to be processed by the computer.
- An input device is necessary for the computer to receive user commands or to receive data to be processed by the processor.

1.2.3 Purpose of output hardware

The following are the purposes of output hardware:

- An output device is connected to a computer to provide data from the computer to the user.
- The output device is controlled by the computer (specifically the CPU).
- Output devices translate machine-friendly outputs into user-understandable outputs.
- An output device is necessary for a computer to share its results and to prompt the user for more information or commands.
- An output device can accept data from other devices and further processes the data accordingly. However, it usually does not transmit data to other devices.

1.2.4 Purpose of processing hardware

Processing is the core functionality of a computer system. The processing stage involves transforming data into information. The term *data* is used to define raw facts, while *information* is used to describe processed data. Depending on the type of motherboard, CPU or RAM, different computer systems will process data at different speeds.

1.2.5 Purpose of storage hardware

Storage is the crucial part of the computer system. Storage hardware is used to store data or instructions before and after processing.

There are two types of storage:

- Primary storage
- Secondary storage.

Primary storage

Primary storage (also known as the *main memory*) is the component of the computer that holds data, programs and instructions that are currently in use. In a computer, the RAM stores data temporarily while it is in use. RAM is volatile, i.e. data stored in it is lost when we switch off the computer or if there is a power failure. RAM serves as the 'working memory' of the computer, holding data generated by programs.

There are two types of RAM:

- Static RAM
- Dynamic RAM.

The read-only memory (ROM) stores the instructions that turn on the computer when a power button is pressed. ROM is a type of memory that can only be read, not written. ROM is non-volatile, i.e. it does not lose instructions when the power is turned off. ROM instructions remain permanently stored. There are different types of ROM, e.g. PROM, EPROM and EEPROM.

Secondary storage

Secondary storage devices can be internal or external storage devices that are non-volatile. Secondary storage can typically store data ranging from a few megabytes to petabytes.

Examples of secondary storage are the following:

- Solid-state drives
- Hard disk drives
- Cloud storage (online storage)
- CD-ROM, DVD drives
- Blu-ray drives
- USB flash drives
- SD cards
- Tape drives.

Cloud storage (online storage)

Backups of businesses are now often done using online storage rather than tapes or USB flash drives. Some cloud storage companies offer free storage for individuals and only charge them if their allocated quota is exceeded. Dropbox, Google Drive and Skydrive are some companies that you can use to try out cloud storage.

Blu-ray drives

In recent years, Blu-ray discs have become increasingly popular for film distribution. The disk can store 100 GB of data, while DVDs can only store 4,7 GB. Panasonic, Sony, Disney and Dell have all backed the disk.

Hard disk drives

Hard disks contain rotating magnetic disks with read/write heads that move across the disks as they spin at high speeds to read data or write data. As a direct-access medium, disks allow you to access the file directly by moving the read/write head to the correct location without having to navigate through the rest of the file.

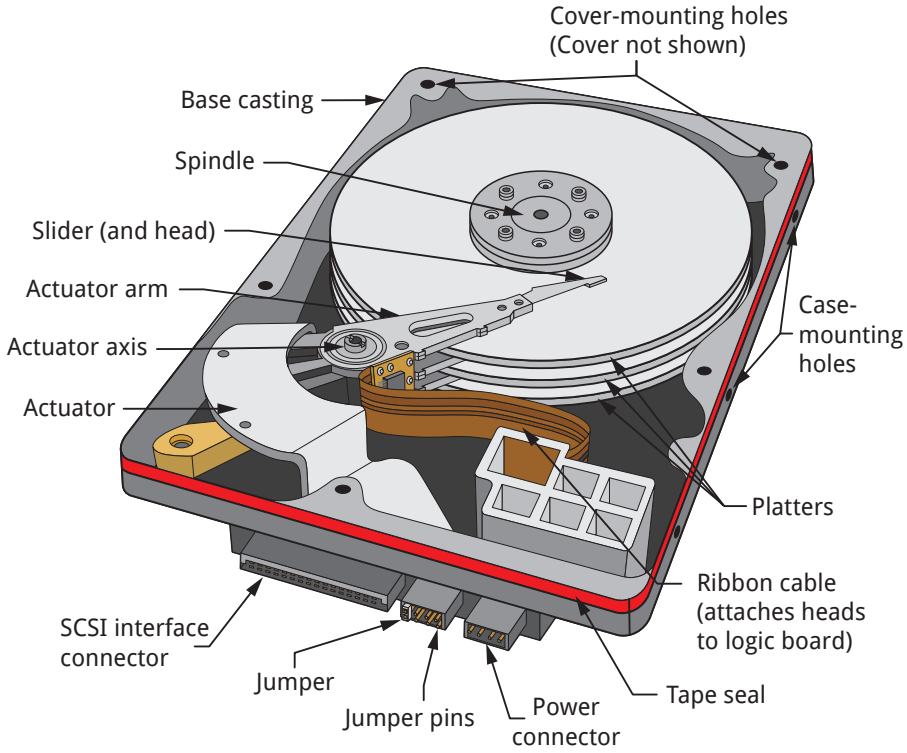


Figure 1.7: Sample hard disk drive

Take a look at the inside part of a computer hard disk drive shown in Figure 1.7. In the read/write head, an electromagnet changes the surface of the disk with a positive or negative charge. This is a representation of binary 1 and 0. The circuit boards carefully coordinate the rotation of the disk and swing of the actuator arm so that the read/write head can access the target location quickly. Hard drives eventually wear out and break due to their moving parts.

1.2.6 Examples of processing hardware

Some of the most common processing devices inside of a computer include the following:

- CPU
- Motherboard
- Network card
- Sound card
- Video card.

The CPU is also known as a *processor*, *central processor* or *microprocessor*. For its power, the CPU is regarded as the brain of the computer system. All computer instructions are processed by the CPU, which works with both the computer's hardware and its software. For example, your computer's CPU processes the instructions necessary to open and display a webpage using a web browser. The major types of CPU are classified as single-core, dual-core, quad-core, hexa-core, octa-core and deca-core processors. The leading manufacturers of CPUs are AMD and Intel. Intel CPUs range from i3s to i7s. Some of the AMD processors include AMD Ryzen 7 5800X3D and AMD Ryzen 9 5900X.

Figure 1.8 shows an image of a CPU.

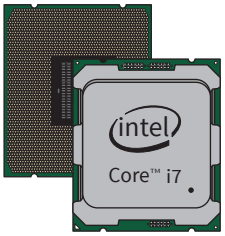


Figure 1.8: Example of an i7 CPU

Motherboards are the largest boards in a computer chassis, as they contain both a printed circuit board and electrical components. A motherboard is also known as a *mainboard*. The processor interacts with the CPU, RAM and the rest of the computer's hardware to manage electricity. Figure 1.9 shows an example of a motherboard.

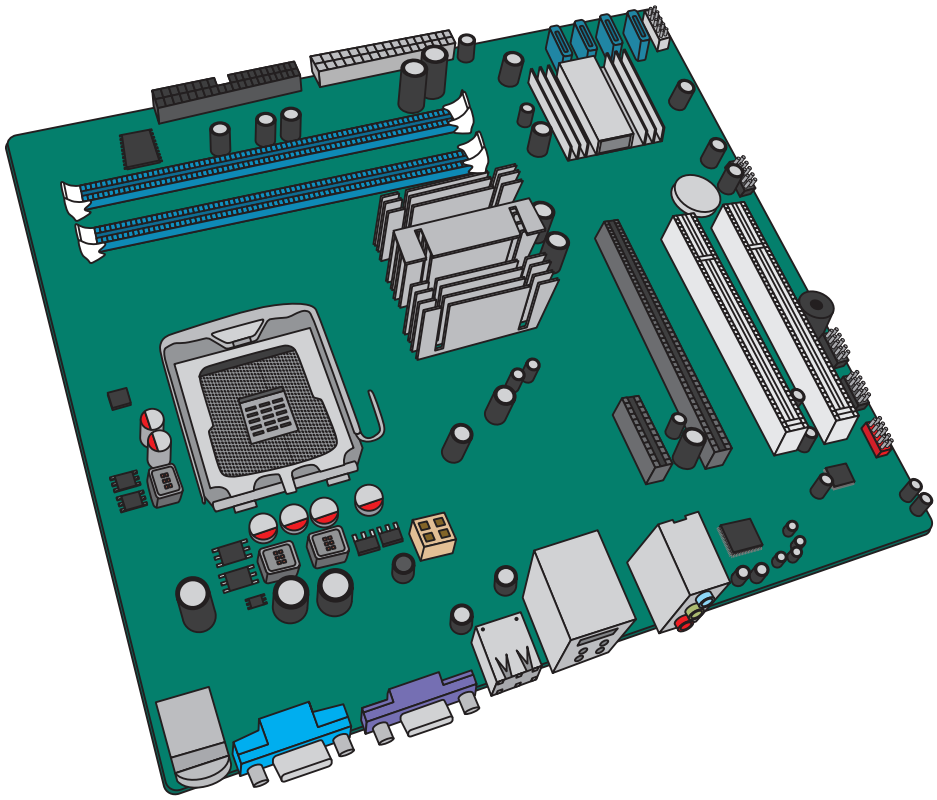


Figure 1.9: Acer desktop motherboard (G41)

Sound card

Computer sound cards are hardware components installed on the motherboard that provide audio input and output capabilities. Mostly, sound cards have at least two analogue line input connections and one stereo line output connection.

Video card

A video card, often known as a *display adapter* or *graphics card*, is a computer expansion card that plugs into the motherboard. You wouldn't be able to see the output on the computer display without one. Because video cards have more processing power and video RAM, gamers prefer them to integrated graphics.

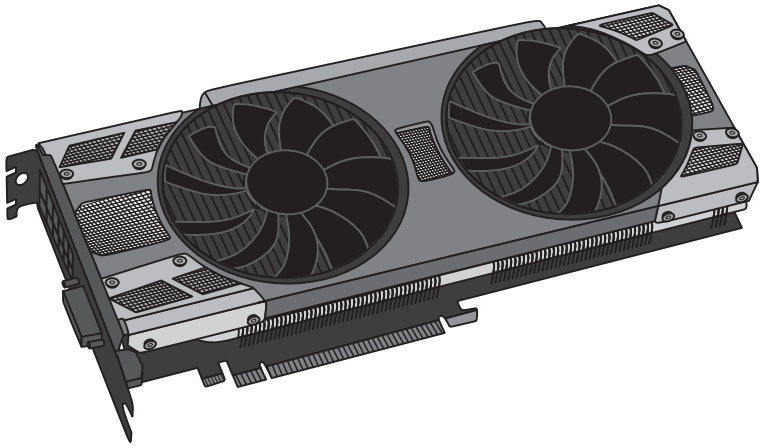


Figure 1.10: Video card

1.2.7 Output hardware

Hardware components that disseminate and display both data and information are classified as *output devices*. Any information processed by and sent out from a computer or other electronic device is considered output. Output can be in the form of text, visual, audio or hardcopy. One distinction that can be drawn between output devices is that of hardcopy versus softcopy. Hardcopy devices (mostly printers) produce a tangible and permanent output, whereas softcopy devices (display screens) present a temporary, fleeting image. The most common examples of output devices are monitors, printers, plotters, projectors, speakers, headphones, etc.

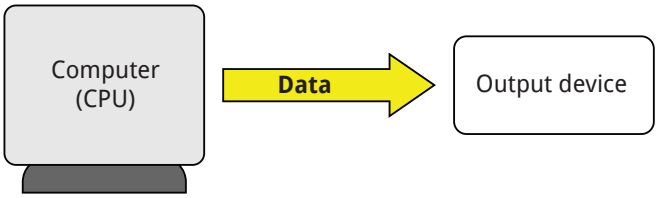


Figure 1.11: Relationship between input device and the computer

Examples of output devices explained include monitors, touchscreens, projectors, speakers, printers and plotters.

Monitors

The most common example of an output device is a monitor. A monitor is also known as *visual display unit*. The monitor provides output in symbolic and graphical form, generated using small dots known as *pixels* or *megapixels*. The more pixels a monitor has, the better the sharpness and quality of output. Monitors are broadly classified into two types, namely cathode-ray tubes and flat-panel displays.

- Cathode-ray tubes monitors – These contain millions of phosphorous dots in red, green and blue. The smaller the pixels, the better the quality of the screen. Cathode-ray tube monitors consume a lot of power in comparison to flat-panel displays. For this reason, companies are producing flat panels only.
- Flat-panel display monitors – These can be further split into emissive and non-emissive display. Emissive displays use optical effects to transform

light from other sources into graphics patterns. Liquid crystal display (LCD) panels and plasma panels are good examples of emissive displays. This technology is mainly used on laptops, monitors, televisions and smartphones. Plasma monitors are relatively thinner and brighter than LCDs.

Touchscreens

A touchscreen is a display screen that also serves as an input device. A computer user interacts with the screen by touching pictures and words on the screen.

Projectors

An output device, such as a projector, transfers images from a computer to a wall, surface or projection screen. The modern projector uses lasers to directly project images.

Speakers

A signal is sent to the computer speakers from a sound card. The internal amplifier vibrates at different frequencies in order to adjust the volume. Headphones are also another form of speakers.

Printers

The most common function of a printer is to produce hardcopies of the processed data. Printers are broadly classified into two types, namely impact printers and non-impact printers.

IMPACT PRINTERS	NON-IMPACT PRINTERS
Print characters and graphics by striking the paper	Print characters and graphics on a piece of paper without striking it
Electromechanical devices are used	No electromechanical device is used
Have banging noise of needle on paper	Are more silent compared to impact
Are slightly cheaper than non-impact printers	Are more expensive
Examples: daisy-wheel, dot matrix	Examples: inkjet, thermal, laser printers

Table 1.3: Characteristics of printers

Plotters

A plotter is a device that is almost identical to a printer. A plotter receives commands from the computer and then draws its picture on the page. Plotters can print on fabric, cardboard film and other synthetic materials. Figure 1.12 shows a picture of a plotter.

NOTE

It is important to note that some devices work as both input and output devices. They can accept input as well as produce output. Touchscreens, interactive whiteboards, digital cameras, pen drives, digital cameras and headsets with microphones are a few examples. See Figure 1.13.

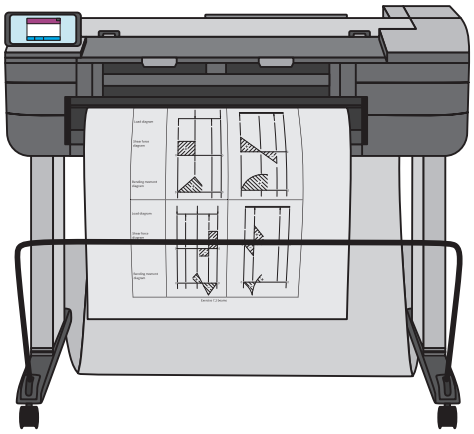


Figure 1.12: Plotter

INPUT DEVICES	BOTH	OUTPUT DEVICES
Barcode scanner	CD/DVD	Braille reader
Joystick	Digital camera	Printer
Scanner	Modem	GPS
Light pen	Pen drive	Plotter
Mouse	Touchscreen	Monitor
Keyboard		Speakers
Stylus		Projector
Microphone		Headphones
Trackpad		Video card
QR code scanner		Sound card

Figure 1.13: Summary of hardware devices

Task 1.1

List the different examples of impact printers and non-impact printers. Use the internet to search for your answers. You can write short notes for yourself.

1.2.8 Typical components of the system unit

The system unit is the computer component that contains the essential devices necessary to do computations and generate results. This system is made up of the motherboard, CPU, RAM and other components, along with the casing that protects the sensitive electrical components from external effects. Most housings are made of steel or aluminium, but they can also be made of plastic. A system unit is sometimes called a *computer tower* or a *computer case*.

In section 1.2.6, we described what motherboards and CPUs are. In this section, we will explain in brief how the CPU operates.

A CPU has the following components:

- Control unit – controls all parts of the computer system. It manages the four basic operations of the fetch-execute-cycle as follows:
  - Fetch – gets the next program command from the computer's memory
  - Decode – deciphers what the program is telling the computer to do
  - Execute – carries out the requested action
  - Store – saves the results to a register or memory

- Arithmetic logic unit – performs arithmetic and logical operations
- Register – saves the most frequently used instructions and data.

Figure 1.14 illustrates the fetch-decode-execute cycle.

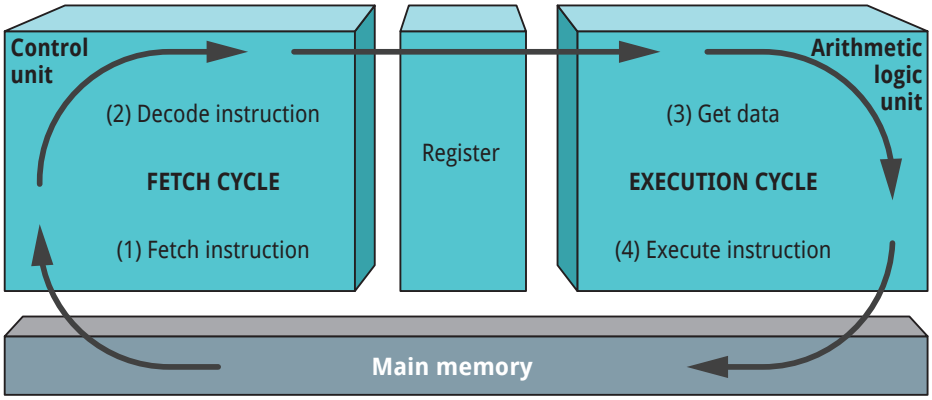


Figure 1.14: Fetch-decode-execute cycle

1.2.9 Modular design of a computer-based system

A modular computer system is made up of easily replaceable parts and standardised interfaces. Computer users can easily upgrade specific aspects of their computers without purchasing a brand-new one, as long as the components use the same standard interface. Computers are an excellent example of **modular design**. The term *modular* can be applied to both hardware and software. Figure 1.15 depicts the modules of a computer.

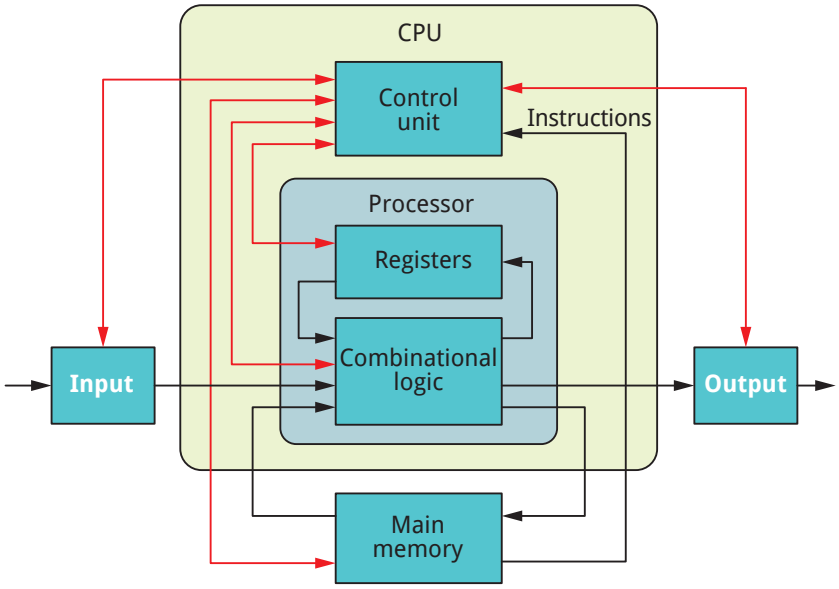


Figure 1.15: Modules of a computer system

VOCABULARY

Modular design – *design principle that divides a system into smaller parts known as modules that can be created, modified, replaced or exchanged independently with other modules or between different systems*

1.2.10 Single-board computer

Single-board computers (SBCs) are fully functional computers in which the microprocessor, input/output functions, memory and other features are all integrated into a single circuit board, with a predetermined amount of built-in RAM and no expansion slots. Some SBCs can be partially expanded or reconfigured, while others are fixed. The most common applications for SBCs are in industrial settings, where they are rack-mounted for control or embedded in other equipment for control.

Advantages of SBCs

- Compact form and portable
- Rich flexible I/O ports
- Durability and longevity
- Good performance at low price
- Relatively low power consumption
- Tremendous community support available for most boards.

Disadvantages of SBCs

- Price – When it comes to high-capacity items, it is more cost-effective to conduct your own design and validate engineering costs.
- Flexibility – A system-on-a-chip is more important to consider than a system board computer, especially if you want a lot of customisations.
- Usually quite fragile (electrically).
- The performance of these boards does pale in comparison to modern consumer desktops or laptops.

1.2.11 Examples of single-board computers

- Raspberry Pi
- Odyssey STM32MP157C
- Rock Pi 4 Model B
- Coral Dev Board
- Seeduino Cloud
- Hikey970 development board
- NVIDIA Jetson Nano developer kit.

1.2.12 Purpose and uses of single-board computers

SBCs are frequently employed in embedded applications. An embedded computer cannot be expanded upon and contains only the input and output capabilities it needs for the task for which it is designed. For example, a vending machine might have an embedded SBC in it to control the functions of the vending machine, but there would be no provision to add more hardware to the computer to expand its capabilities. The applications of this technology include consumer, industrial, automotive, medical, commercial and military uses.

Often, SBCs are plugged into a backplane. The backplane connects input and output devices to the computer. SBCs are frequently used in rack systems, which allows for reliable and fast integration into a system.

1.2.13 Components of a single-board computer

SBC architecture consists of a processor core, memory banks, buses and peripherals. These components are connected by an internal data bus. SBC platforms now feature multiple cores over 1 GHz as a standard feature. Figure 1.16 shows the general architecture of an SBC.

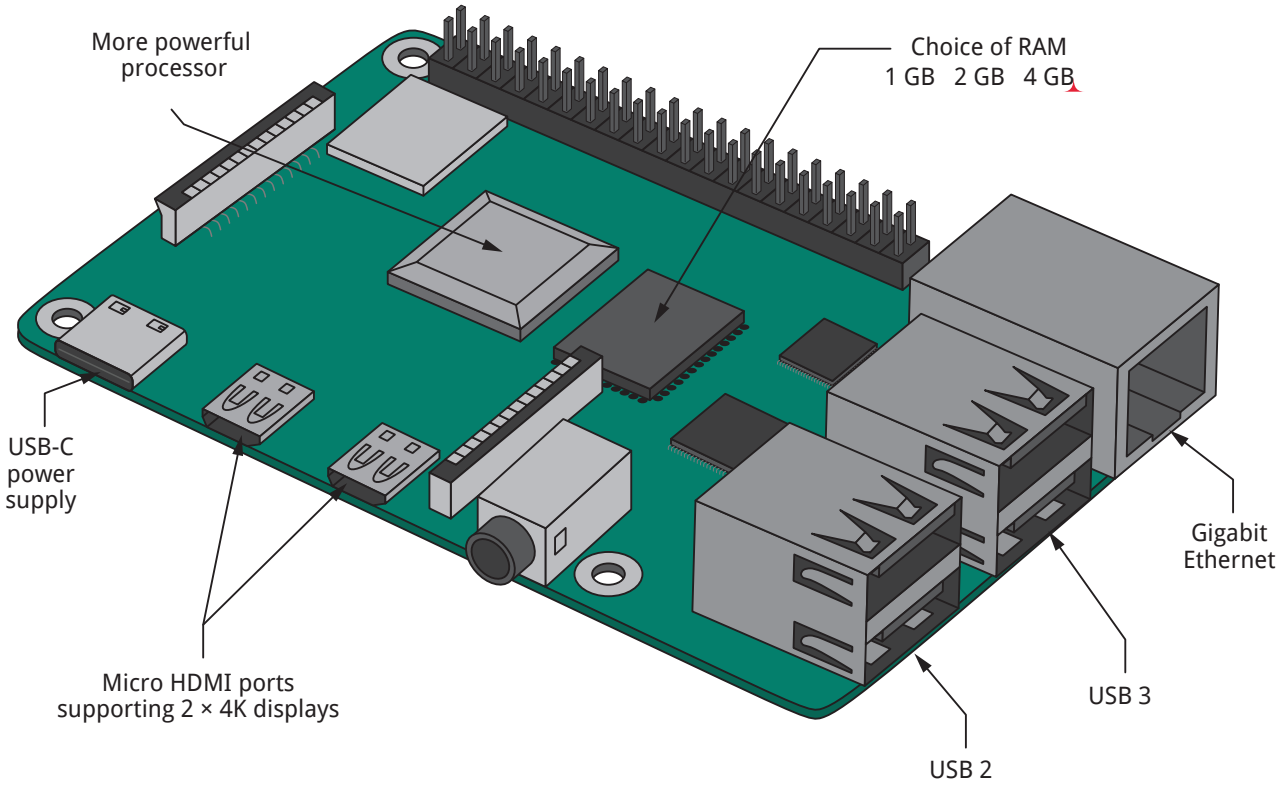


Figure 1.16: Components of an SBC
(Source: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/)

eLINKS

Visit this link to learn more about computer hardware:

- Part 1: ~~https://youtu.be/8UyJMiYqvs4~~
- Part 2: ~~https://youtu.be/gaN1SKti3ts~~

Activity 1.3 **INDIVIDUAL ACTIVITY**

1. Define the following terms as applied in computer hardware:
   - **1.1** Input hardware
   - **1.2** Processing hardware
   - **1.3** Storage hardware
   - **1.4** Output hardware (2 × 4) (8)
2. Explain the difference between *softcopy devices* and *hardcopy devices*. (2)

Activity 1.3 (continued)

3. Choose a term from COLUMN B that matches a description in COLUMN A. Write only the letter next to the question number. (7 × 1) (7)

COLUMN A		COLUMN B	
3.1	A storage device used with a card reader	**A**	resolution
3.2	A camera on the computer that captures media such as pictures and videos mostly	**B**	cloud
3.3	Enters information such as letters and numbers into a computer	**C**	SD card
3.4	Converts electronic data into a hardcopy	**D**	webcam
3.5	A commonly used portable storage device	**E**	fax machine
3.6	Measured in megapixels; the higher, the better quality	**F**	printer
3.7	A type of storage accessible anywhere in the world as long as there is Internet	**G**	flash disk
		H	keyboard
		I	scanner

4. What does the acronym *SBC* stand for? (1)

5. List FIVE examples of SBCs. (5)

6. State whether the following statements are true or false. Correct the statement if it is false.

6.1 A scanner captures media, such as pictures, videos and sound.

6.2 A fingerprint scanner scans documents, such as photographs and pages of text, and converts them into a digital format.

6.3 A monitor displays or sends data from a computerised device to other users.

6.4 Speakers convert data on a computer to sound.

6.5 A digital camera is both an input and an output device.

6.6 Cloud storage is for free for a specific amount. (6 × 1) (6)

7. *CPU* stands for _____. (1)

TOTAL: 30

1.3 The Windows Command prompt

This unit is meant to give you an overview of the most useful features of the shell. Furthermore, it will provide an overview of useful Windows commands. The commands will vary depending on the operating system you're using. However, some similarities may also exist. In this unit, we will cover most of the basic fundamentals for common operations.

1.3.1 Shells

A **shell** acts as a protective 'wrapper' around the operating system. It protects the user from the system (and from you!). Computers can be used without understanding the low-level details that programmers do. Figure 1.17 shows someone using a shell to tell the computer to copy some files. A number of modern shells are available, and they all function in essentially the same way.

Instruction say to move to previous page – not possible. Not enough space

VOCABULARY

Shell – *program that presents a command line interface that enables you to interact with your computer by entering commands via a keyboard; exposes the services of an operating system to a human user or other programs*

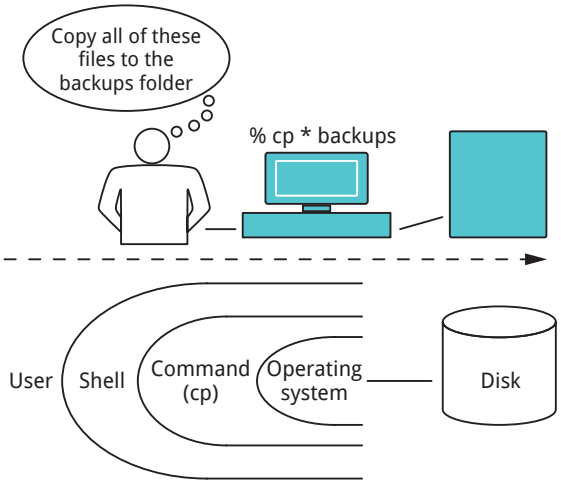


Figure 1.17: The shell interface (Source: Miskovic, 2014)

There are many features in shells that make interaction with a computer faster and easier (in conjunction with utility programs) , including the following:

- Locate files according to their characteristics in the file system of the computer, then perform various operations on them.
- Remember and redo a command, or a series of commands, that you have done before.
- Allows you to jump from one 'memorised' location to another.
- Set up a series of file names or determine how often to perform a task.

1.3.2 Launch a new Command prompt in Windows

The Command prompt has been around for a long time, and it's still a useful tool to have. In this section, we'll teach you how to open the Command prompt in a variety of ways. When you first turn on your computer, some unusual information will flash by. This information is provided by MSDOS to inform you of how your computer is set up. You may disregard it for the time being. When the data has finished scrolling, you will see: `C:\>`. This is known as the Command prompt or the *DOS prompt*. The blinking underscore next to the Command prompt represents the cursor. Options 1 to 4 illustrate the different ways of opening a Command prompt.

Option 1

Open Command prompt by pressing the Windows key and type the terminal and press the ENTER key. You will be taken to the Command prompt as shown.

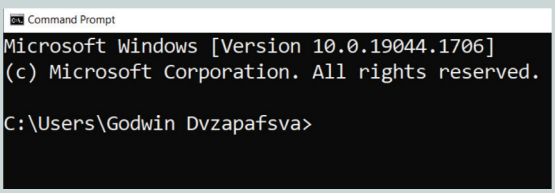


Figure 1.18: Using Windows key

Option 2

Open a Command prompt in Admin Mode from Task Manager the secret, easy way. In the search area, type Task Manager and select FILE and type `cmd`.

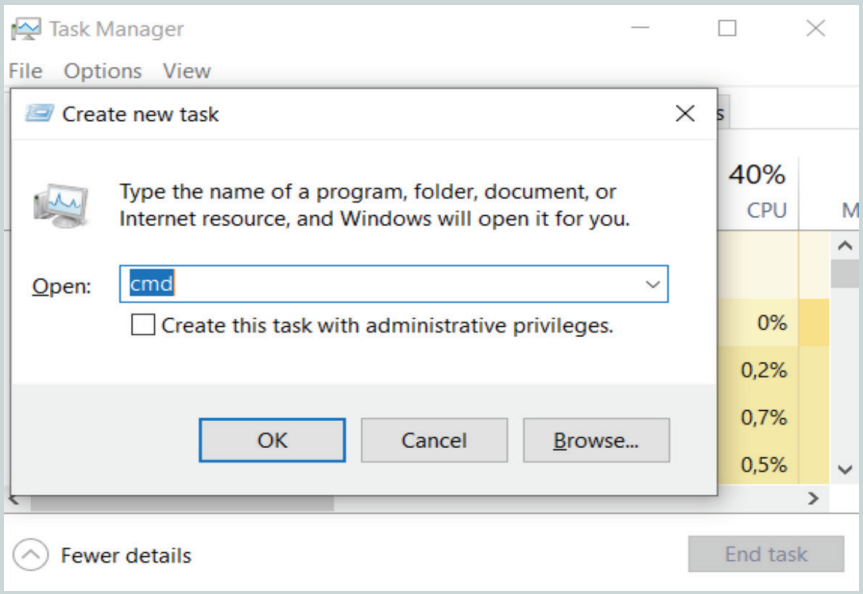


Figure 1.19: Using task manager

The above command takes you to the following path:

```
C:\Windows\system32>
```

Take note of the different path as before. This means you opened Command prompt with administrative privileges – no need to type anything. You can navigate to any path and we will see this in the following section.

Option 3

Open Command prompt from the START MENU search.

By clicking START and typing `cmd` into the search box, you can easily access the Command prompt. By right-clicking the result and selecting Run as Administrator, you can launch Command prompt with administrative privileges. You can also select the result with the arrow keys and press the CTRL + SHIFT + ENTER key combo.

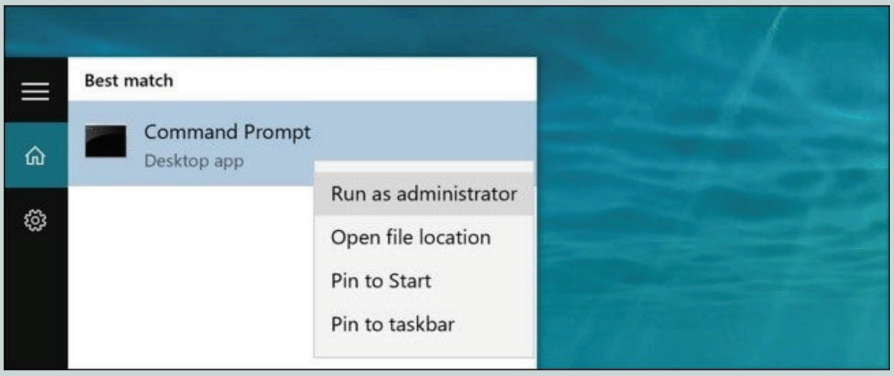


Figure 1.20: Using the Start menu

Option 4

Open Command prompt from the File Explorer Address bar.

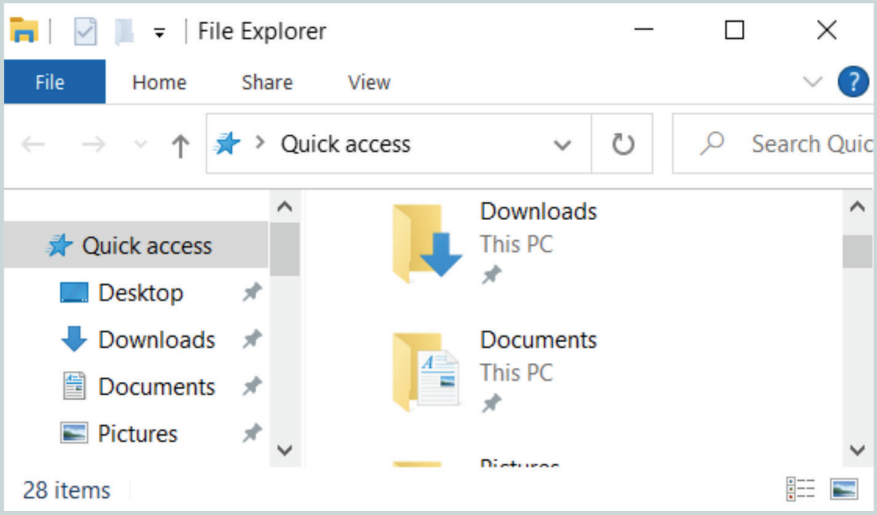


Figure 1.21: Using File Explorer

In File Explorer, click the address bar to select it (or press ALT + D). You can open the Command prompt by typing `cmd` into the address bar and pressing ENTER to automatically set the current folder's path.

1.3.3 Use the `help` command to list some common commands

`help` is a Command prompt command that displays additional information about another command. At any time, you can use the `help` command to find out how a command is used and how its many options can be used, including which options are available. The `help` command is available from within the Command prompt in all Windows operating systems. If used without parameters, `help` lists and briefly describes every system command.

Syntax:

```
help [<command>] [/?]
```

ITEM	EXPLANATION
Help	The `help` command is executed without options to display a list of commands that can be used with the `help` command.
Command	The command for which you want help information can be specified here.
/?	The help switch is the /? option that provides help information about a command.

Switch can be used as follows:

```
C:\ > cd /?
```

The output of the above command will look as shown on Figure 1.22:

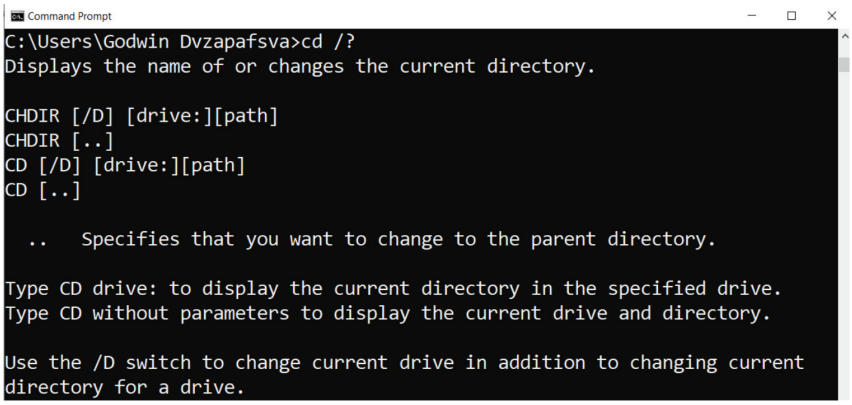


Figure 1.22: Output on Command line

The above example shows what the help switch for `cd` (change directory) can do.

1.3.4 Expand a Windows file path and explain each element

A standard disk operating system (DOS) path can consist of three components:

- *Volume* or *drive letter* followed by the volume separator (:)
- *Directory name* – sub-directories in nested directories are separated by the directory separator (\)
- *Optional file name* – file path and file name are separated by the directory separator character.

Figure 1.23 shows a typical file path.

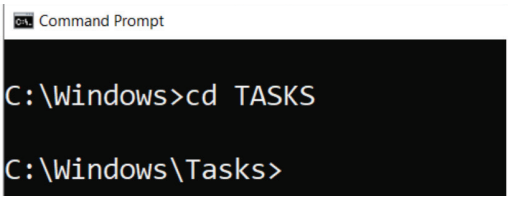


Figure 1.23: File path

The volume or drive letter in the example above is C. The directory is Windows. The drive letter and directory windows are separated by the directory separator symbol (\). Tasks is the sub-directory. You can point to a file name if you want to have it opened or executed.

The route is absolute if all three components are present. If no volume or drive letter is supplied and the directory name starts with the directory separator character, the path is relative to the current drive's root.

1.3.5 List the contents of the current folder using the `dir` command

In this section, you will view the contents of a directory by using the `dir` command. The `dir` command stands for *directory*. To view the contents of a directory, type the following at the Command prompt: `dir`.

A list similar to the following appears:

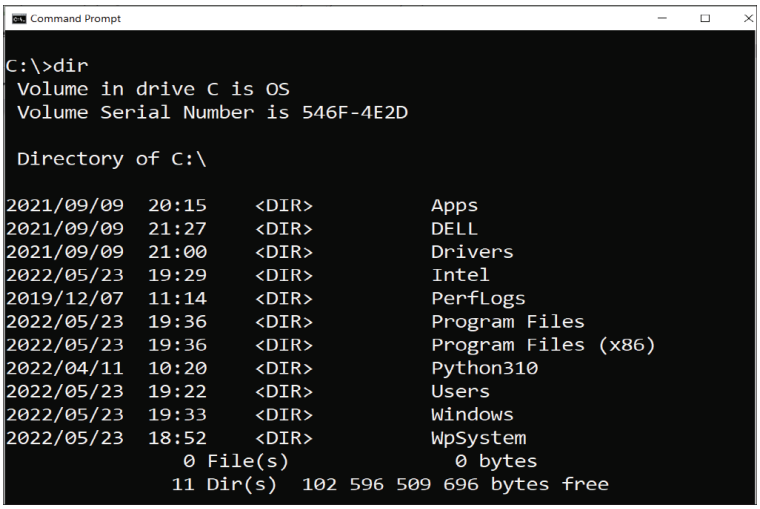


Figure 1.24: Contents of a folder

Figure 1.24 depicts a list of directories. A directory list is a list of all the files and sub-directories included within a directory. In this situation, you may view all of your hard drive's files and folders in the primary or root directory. The root directory (typically the C drive) stores all of your disk's files and folders. In rare cases, there may be several folders that cannot fit on a single page. By entering `c:\>dir/p` at the Command prompt, you may change the `dir` command to enable users to read only a single page. You can also use the /w switch to display a wider page.

1.3.6 Change directory location using the `cd` command

Use the `cd` command to move from your current directory to another. Ensure that you have access to the destination directory. Suppose you are in the following directory:

```
C:\Users\Godwin Dvzapafsva>
```

Put the cursor after the > sign and type `cd documents`, then press ENTER. The result will be as follows:

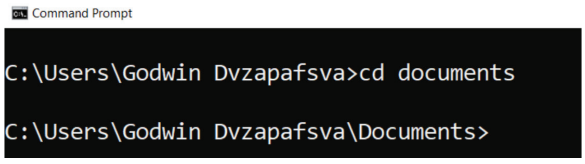


Figure 1.25: Changing directory

To move back to the previous directory, you need to type `cd..` from the current directory, which in the case above is documents. By typing `cd/`, you will be taken to the root directory or `C:\>` prompt.

1.3.7 Create a new folder using the `mkdir` command

`mkdir` is the command you will use to create a directory. Create a folder called NCV in the root directory using the `mkdir` command. Navigate to `C:/>` in order to do so.

NOTE

This command is the same as the `md` *command.*

```
C:\>mkdir NCV

C:\>dir
 Volume in drive C is OS
 Volume Serial Number is 546F-4E2D

 Directory of C:\

2021/09/09  20:15    <DIR>          Apps
2021/09/09  21:27    <DIR>          DELL
2021/09/09  21:00    <DIR>          Drivers
2022/05/23  19:29    <DIR>          Intel
2022/05/23  21:58    <DIR>          NCV
2019/12/07  11:14    <DIR>          PerfLogs
2022/05/23  19:36    <DIR>          Program Files
```

Figure 1.26: Creating a new folder

Task 1.2

Move into the directory called NCV and try creating another sub-directory called Programming using the `md` command.

1.3.8 Remove a folder using the `rmdir` command

To simplify your directory structure, you may want to remove directories you are no longer using. For our practice, we will delete the NCV directory. Navigate to the C:\> directory in which the NCV subfolder is located. Type the `rmdir [directory name]` or `rd` commands to remove a directory. The `rmdir` command stands for *remove directory*. The directory will be deleted with its files and sub-directories.

1.3.9 Remove a file using the `del` command

In this section we describe how to delete or remove a file you no longer need from your disk. In order to free up disk space, you need to delete files you no longer use. The `DEL` command is used to remove a file. The command `DEL` stands for *delete*. For the purpose of practice, recreate the NCV directory and create a Word document called Test.docx inside the folder. Navigate to NCV path and run `del test.docx`, as shown below.

```
C:\NCV>del test.docx

C:\NCV>dir
 Volume in drive C is OS
 Volume Serial Number is 546F-4E2D

 Directory of C:\NCV

2022/05/23  22:31    <DIR>          .
2022/05/23  22:31    <DIR>          ..
               0 File(s)              0 bytes
               2 Dir(s)  102 580 727 808 bytes free
```

Figure 1.27: Deleting a file

NOTE

From C:\NCV you create your Word document by the following command:
type `nul >Test.docx`

Once you delete a file using the `del` *command, you will not find it in the Recycle bin.*
You can also use the keyword **echo** *to enter text into that same file as follows:*

```
C:\NCV>echo "Hello friends. How are you?">Text.docx
```

To add a second line you will need to use two >> signs instead of one. If you use one, the text will be overwritten.

1.3.10 Rename a file using the `rename` command

In this section, you will rename the Test.TXT file you created. Make sure your Command prompt looks like the following: C:\NCV>.

To rename the Test.docx file to Test1.TXT, type the following at the Command prompt:

```
ren Test.exe Test1.txt
```

1.3.11 Copy a file using the `copy` command

Using the command line has the following advantages over traditional methods of file copying:

- It eliminates, for example, the need to copy and paste manually.
- Unattended copying is also possible. That is, after entering specific commands, you can leave the computer alone and take a rest.

Syntax:

```
copy source [/A | /B] [destination]
```

- [/A | /B] are optional.
- /A: Indicates an ASCII text file.
- /B: Indicates a binary file.

To copy a file named *Test1.txt* from the current drive called *Programming* to an existing directory named *NCV* that is located on drive C, type:

```
C:\NCV\Programming>copy test1.txt c:\NCV
```

You can also copy all files in the current directory to the a specific path, e.g. `copy *.* d`:

The above syntax copies the files into the D drive.

If you want to make *test1.txt* a hidden file, you can use command attrib + h test1.txt. To unhide, you do exactly the same, but use the – sign on the *h* property.

1.3.12 Clear the Command prompt screen using the `cls` command

To clear the Command prompt screen, just type the `cls` command from any prompt directory path.

1.3.13 Run an executable file from the command line

You can run EXE files from the Windows command line if you point it to the correct location and specify the correct EXE file name. Navigate to the path and type the executable name with the extension.

EXAMPLE 1.4

Find the file path. Open the folder containing your program, then copy or write down the path in the address bar at the top of the window.

For example, if you want to open Microsoft Word, the executable to run will be in a folder named Office16. In this example, the path would be:

```
C:\Program Files\Microsoft Office\Office16>
```

Type `winword.exe` and press `ENTER`. WinWord is the executable for Microsoft Word.

1.3.14 List the system info using the `systeminfo` command

`systeminfo` provides extensive configuration data about a machine and its operating system, such as operating system configuration, security information, product ID and hardware attributes (such as RAM, disk space and network cards). Type `systeminfo` from the prompt to display all information.

1.3.15 Copy the path of any file by dropping the file in the Command prompt

When you wish to shift directories (e.g. `cd` to a folder deep on your file system but you're staring at that folder on your desktop), enter `cd`, then drag and drop that folder into your command line and press `ENTER`.

Activity 1.4 **INDIVIDUAL ACTIVITY**

You are required to use command lines to do the following requirements:

1. Create a folder called *NCVL2* in the C drive using commands. (2)
2. Add a file called *programming.txt* inside the NCVL2 folder. Add the following text:
   `"Hello colleagues"` into the text file using commands only. (2)
3. Add another line with the following text:
   `"This is my first year learning programming"` (1)
4. Create a folder called *Practice* inside NCVL2. (2)
5. Move the text file Programming.txt into the folder Practice. (2)
6. Show the contents of Practice. (2)
7. Clear all text on the Command prompts. (1)
8. Set the programming file to hidden property. (2)

TOTAL: 14

eLINK

Visit this link to learn more about Windows command line commands:
https://github.com/security-cheatsheet/cmd-command-cheat-sheet

Summative assessment

1. Define the term *hardware* in relation to computers. (2)
2. What is the difference between *analogue signals* and *digital signals*? (4)
3. Classify the following hardware devices into the different categories by completing the table below. (20)
   Use the following words:

   keyboard; CPU; RAM; modem; plotters; barcode reader; arithmetic logic unit; register; image scanner; sound card; webcam; light pen; ROM; monitor; projector; disks; USB; headsets; digital camera; printers; SSD; speakers; touchscreen; pen drive

INPUT	PROCESSING	OUTPUT	STORAGE	INPUT/ OUTPUT

4. List the THREE main components of a CPU. (3)
5. List the TWO categories of computer memory. (2)
6. What is a *system unit*? (2)
7. Define the term *modular design* in relation to computer-based systems. (2)
8. What is meant by the term *single-board computer*? (2)
9. Identify FOUR advantages and THREE disadvantages of single-board computers. (8)
10. Information that comes from external source and fed into computer software is called . . .
    - **A** throughput.
    - **B** reports.
    - **C** input.
    - **D** output. (1)
11. Which of the following is used to hold ROM, RAM, CPU and expansion cards?
    - **A** Cache memory
    - **B** Computer bus
    - **C** Motherboard
    - **D** All of the above (1)

Summative assessment (continued)

12. The CPU is the … of the computer.

A brain
B eye
C ear
D All of the above (1)

13. Choose a term from COLUMN B that matches a description in COLUMN A. Write only the letter next to the question number. (8)

COLUMN A		COLUMN B	
14.1	Used to delete one or more files	**A**	cls
14.2	Displays a list of files and folders contained inside the folder in which you are currently working	**B**	rmdir
14.3	Copies one or more files from one directory to another	**C**	systeminfo
14.4	Provides help information for Windows commands	**D**	time
14.5	Used to create a new folder	**E**	dir
14.6	Used to delete an existing or completely empty folder	**F**	copy
14.7	Used to display basic Windows configuration information for the local or a remote computer	**G**	init
14.8	Used to clear the screen or console window of all previously entered commands and any output generated by them	**H**	del
		I	mkdir
		J	ls
		K	help

14. What is a *root directory*? (1)

TOTAL: 57

Module 2

Problem solving in computer programming

After you have completed this module, you should be able to:

- define the term *problem solving*;
- define the term *calculational thinking*;
- describe the phases of a program development life cycle;
- describe the purpose of problem solving leading to solutions;
- explain and apply various problem solving steps;
- use appropriate tools and techniques to present a solution;
- define the term *algorithm* and its purpose in the problem solving process;
- list examples of algorithms in life;
- construct and devise an algorithm;
- describe the purpose of input, processing and output as part of the solution-creation process;
- define the term *IPO chart* and explain its purpose in solution development;
- outline the various symbols used as part of a *flow chart*;
- explore and create algorithms in the form of an IPO chart and a flow chart that include various constructs.

Introduction

Problem solving is the core of computer science. In this course, you will be taught how to solve problems using a programming language. We will examine how a person or a business solves an issue, and how a computer can implement this design using a *syntax* that it understands – this refers to the rules that define the combinations of symbols used in the programming language.

Programming is the process of creating a set of instructions that tell a computer how to perform a task. Machine code is difficult for programmers to learn, program in and debug. As a result, most programmers create code in high-level programming languages. These languages are close to the programmer's spoken and written language. Python, for example, forms instructions using 'print', 'if', 'input' and 'while' expressions, all of which are English terms. Instructions sometimes resemble shortened English sentences.

The job of a computer programmer is to solve problems. The following are required to do this:

1. Understand how to represent the information (data) describing the problem.
2. Identify the steps for transforming the information.

NOTE

Programming can be done using a variety of computer 'languages', such as Java, Python, C++, C#, Go, Kotlin, etc. A program instruction written in a high-level language is called source code. Computers cannot understand and execute source code unless it is translated into machine code. This is achieved through compilers or interpreters.

2.1 Problem solving process and concepts

2.1.1 Definition of problem solving

In general terms, **problem solving** refers to the process of finding solutions to difficult or complex issues. A computer is not a clever machine that can find solutions when problems arise. It really recognises only a few numbers and characters or lists (called *arrays*) of these items. It needs humans to solve the problem and then translate it into something the computer understands.

2.1.2 Definition of calculational thinking

Calculational thinking refers to methods for solving problems by expressing them in a way that can be understood by a computer. It involves automating processes, as well as using computing to explore, analyse and understand processes (natural as well as artificial).

VOCABULARY

Problem solving – *defining the problem, determining its root cause, identifying, prioritising and deciding on options, as well as implementing the solution*

Calculational thinking – *interrelated set of problem solving methods that involve expressing problems and their solutions in ways a computer can also execute*

Calculational thinking can be divided into the following four steps, regardless of whether it is applied in computer science or another subject area.

- Decomposition – to address a complicated problem, you must first divide it into smaller, more manageable sections
- Abstraction – the process of finding the most relevant information from a larger data set; helps to define/generalise what has to be done to tackle the problem as a whole
- Analysis – solution execution and evaluation
- Algorithmic thinking – creating a logical, step-by-step blueprint for solving a problem; repeatable for a predictable, consistent outcome.

2.1.3 The phases of the program development life cycle

The process of creating application programs is called *program development*. A *program development life cycle* (PDLC) is a systematic approach to developing quality software. Each step of program development must be successfully completed before the next step can be taken. PDLC contains the following five phases of program development: analysis, design, coding, debugging and testing, and implementing and maintaining application software.

VOCABULARY

Syntax – *rules that define the combinations of symbols that a computer's processor can interpret*

Syntax refers to the 'spelling and grammar' of a programming language. Computers are inflexible machines that understand what you type only if you use the exact form that the computer expects. The expected form is called the *syntax*. A program with syntax errors cannot execute.

A *logic error* (or logical error) is a mistake in a program's source code. This results in incorrect or unexpected behaviour. It is a type of runtime error that may simply produce the wrong output or may cause a program to crash while it is running. Many different types of programming mistakes can cause logic errors.

Seven steps in the PDLC

The figure below shows the seven steps of a PDLC.

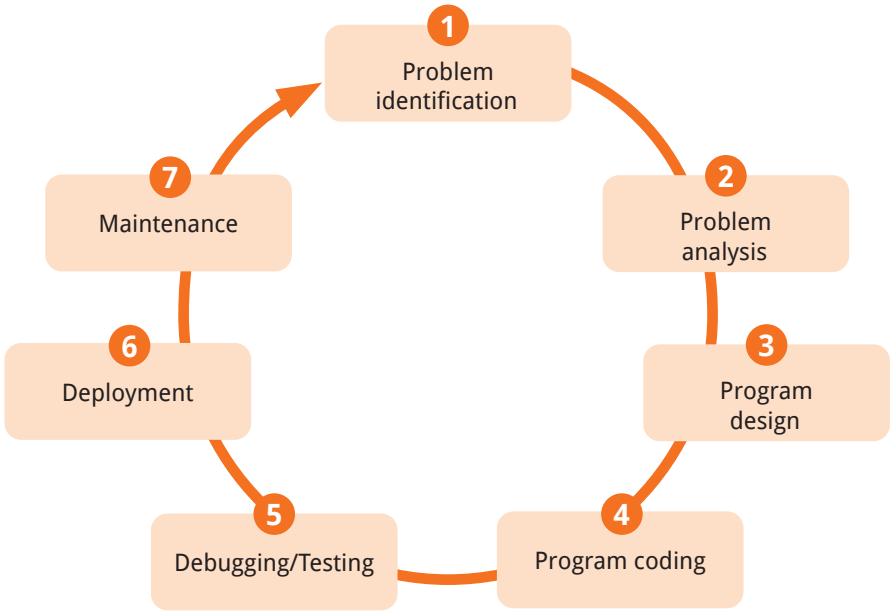


Figure 2.1: PDLC

1 Problem identification

Defining the problem is the first step. System analysts are responsible for providing programmers with the results of their work in the form of a program specification for major software projects. Program specifications include the data used by the program, the processing to find the solution as well as the output format and user interface.

2 Problem analysis

The computer user must figure out the problem and then decide how to resolve it. In most cases, system analysts will try to break down the problem into smaller manageable tasks that are easier for the programmer to understand. The software requirements document is created once the analysis and expectations have been specified. It includes all the requirements of the project as specified by the customer. This document forms the basis for all program development. Once accepted by the customer, further development can take place. One of the options that can be used is a simple input-process-output (IPO) chart or table – refer to section 2.2.6 where this term is explained in more detail.

3 Program design

The development team selects the best or ideal option after investigation. In this phase, the software design document or design document specification is prepared as per the software requirements document. During the design phase, you concentrate on the core objective that the program is attempting to achieve, and then you identify all the elements that contribute to this aim. Modular programming, sometimes known as ***top-down programming***, is a style of programming that starts with a high-level description of what is to be done and this is then broken down into simpler pieces. During this process, a programmer can use tools such as **algorithms** (refer to section 2.1.1), flow charts (refer to section 2.2.8), pseudocodes and decision tables.

VOCABULARY

Top-down programming – *programming style in which the design begins by specifying complex pieces and then dividing them into successively smaller pieces*
Algorithm – *procedure used for solving a problem or performing a calculation; an algorithm acts as an exact list of instructions*

4 Program coding

The high-level design document is now translated into **modules** and code functions. These are distributed among different developers. Developers will write the source code, which will then be translated into machine code. Compilers, interpreters and tools are used to implement the code at this step. Coding is one of the longest phases of the PDLC, because this is where the actual solution is produced.

VOCABULARY

Module – *any of several distinct but interrelated units from which a program may be built up/into which a complex activity may be analysed*

Compilers take the entire source code and translate it into object code at one go. As soon as the object code has been converted, it can be run at any time without assistance. The process is known as *compilation*. Examples of compilers include Microsoft Visual Studio, GNU Compiler Collection (GCC), Common Business Oriented Language (COBOL), etc.

Advantages of compilers

- The whole program is validated so there are no system errors.
- The executable file is enhanced by the compiler, so it runs faster.
- The user does not have to run the program on the same machine on which it was created.

Disadvantages of compilers

- Compilers do not usually spot errors.
- The source code compiled is platform-dependent.
- It requires more space for source code and generated object code.

Interpreters translate source code into object code one instruction at a time. It is the equivalent of a human interpreter who translates what is being said into another language directly, i.e. while the speaker is addressing the audience. The translated object code is then executed. This process is known as *interpretation*. Examples of interpreters include OCaml, List Processing (LISP), Python, etc.

Advantages of interpreters

- Programs can be run before they are complete to get partial results immediately.
- They require less available memory.
- If you discover an error before you complete your program, you can learn from it.

Disadvantages of interpreters

- Unverified scripts may contain syntax errors.
- Because each execution is interpreted, it may be slow.
- No executable file is produced. The source code program must therefore be provided, and it could be altered without permission.

Assemblers translate assembly language into object code. In contrast to compilers and interpreters, assemblers create one machine code instruction for each assembly instruction. Examples of assemblers include Fortran Assembly Program (FAP), Macro Assembly Program (MAP) and Symbolic Optimal Assembly Program (SOAP).

Advantages of assemblers

- It is easier to fix errors and alter program instructions.
- Like machine-level language, it is executable.
- The symbolic programming is easier to understand.

Disadvantages of assemblers

- It is difficult to maintain.
- The design of a program can be invalidated by a small change.
- It is machine-dependent.

VOCABULARY

Compiler – *special program that translates a programming language's source code into a set of machine-language instructions that can be understood by the computer's processor*

Interpreter – *computer program that directly executes instructions written in a programming language (does not require the program to be converted into machine language)*

Assembler – *program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform basic operations*

5 Debugging/Testing

Software testing is the practice of comparing software to its requirements. Testing is used to assure the quality of the final product. Software testing can only assist in the detection of flaws; it cannot ensure the absence of defects. Finding flaws in the product development cycle sooner rather than later is far more cost-effective. The product is tested for bugs by a group of testers. ***Bugs*** are errors in the program. They need to be corrected. *Debugging* is the process of finding the bugs in the computer program. Several testing levels are described in the Table 2.1.

TESTING LEVEL	DESCRIPTION
Unit testing	Done to verify the functionality of a unit of code
Integration testing	Assesses the interface between software components
System testing	Overall testing of the software system
Regression testing	Ensures that each new fix does not break anything that was previously working
Acceptance testing	Performed to determine whether or not the software system has met the requirement specifications; two forms of acceptance testing are *alpha* and *beta testing* (described below)
Alpha testing	Type of validation testing
Beta testing	Conducted at one or more customer sites by the end user of the software; the version is made available to a limited number of people

Table 2.1: Software testing levels

Once testing is completed, documentation will commence.

6 Deployment

One must run the program to make sure there are no syntax and logic errors. Once the solution has been thoroughly tested for logic and syntax errors, **deployment** will commence. The project manager deploys the product live for real-time use by the client.

7 Maintenance

The focus shifts from development to maintenance when the program is live and in active use. Developers/Maintenance teams should keep the product working at its full potential and ensure that the user has no trouble using it.

VOCABULARY

Bug – *coding error in a computer program*

Deployment – *all the processes involved in getting new software (or hardware) up and running properly*

2.1.4 Purposes of problem solving leading to solutions

The ability to solve problems is important for both individuals and organisations, as it allows us to exercise control over our environment.

Here are some of the reasons for problem solving:

- Fixing things that are broken
- Addressing risk
- Improving performance
- Seizing opportunity.

2.1.5 Problem solving steps

George Polya was a Hungarian teacher and mathematician who lived from 1887 to 1985. In 1945 he published a book called *How To Solve It* in which explains that people can learn to become better problem-solvers by following four simple steps. These steps are set out in this section.

1 Understand the problem

When you first look at a problem, you should carefully read it and check whether you understand it. Ask yourself, "What do I know and what do I want to learn?"

Look at the following example: If $b^2 = a$, then b is the square root of a.

To calculate the square root of a, you must first discover a # that, when squared, equals a. Consider the following: We want to know what integer, when squared, equals 1,444. In this scenario, the challenge may be written as ($b^2 = 1{,}444$).

2 Devise a plan

For the second step, you must devise a strategy for using what you know already. Consider how the problem connects to topics you are familiar with or previous problems you have addressed. This problem can be solved using a guess-and-check (trial-and-error) method or an algebraic square root method. So, how do we calculate the square root? There are some simple values for which you may calculate the square root, such as $\sqrt{25} = 5$, $\sqrt{36} = 6$, ... $\sqrt{81} = 9$.

This approach needs you to begin by guessing and then calculating how far off your answer is. You then update your guess and try again! You want to know what the b is in $b^2 = 1{,}444$.

Strategy: Determine what b is to equal 1,444. ($b = 1{,}4444$).

Apart from trial and error and guessing, other techniques that can be used are the following:

- Look for a pattern
- Make an orderly list
- Draw a picture
- Eliminate possibilities
- Solve a simpler problem
- Use a model
- Consider special cases
- Work backwards
- Use direct reasoning
- Use a formula.

3 Carry out the plan

This is the step during which you put your strategy into action. Now that you have used the guessing and checking approach, put it to use! So, try the following:

30 × 30 = 900
35 × 35 = 1 225
40 × 40 = 1 600

You can now deduce that your answer must be between 35 and 40. So, you can see how you have narrowed the problem down. After more refining, the answer turns out to be 38.

4 Look back

Finally, at this step, you examine and double-check your results. Let us use the original example:

- Have you responded to the original question? Yes, you have addressed the 1,444 = 38 question.
- Is there a method to evaluate if your response is reasonable? Yes, simply multiplying 38 × 38 = 1,444. You may also use a calculator or search for alternatives if they exist.

2.1.6 Using appropriate tools and techniques to present a solution

Defining the problem and then identifying the approach needed for its solution can be difficult. There are several design tools that can help you in the process.

User stories

A user story is a brief, plain-language explanation of one or more features or functions of a software program written from a user's point of view.

VOCABULARY

Sprint – *set amount of time that a development team has to complete a specific amount of work*
Feature – *unit of software that provides a specific function or capability for the software*

User stories:

- can be understood by anyone;
- are bite-sized deliverables that can be completed in **sprints**, unlike full features that are delivered only once the product team has completed all the **features**;
- focus on the real people, not abstract features; and
- give development teams a sense of progress to help drive them.

User stories are typically written according to a similar formula, normally a sentence or two, as illustrated a below:

`As a [description of user], I want [functionality] so that [benefit].`

Example:
```
As a team leader I want to get the functionalities
    of all modules so that the product meets
    requirements.
```

In practice, user stories might look like these:

- As a database administrator, I want to automatically merge datasets from different sources so that I can more easily create reports for my internal customers.

- As a brand manager, I want to get alerts whenever a reseller advertises our products below agreed-upon prices so that I can quickly take action to protect our brand.

In many quick-thinking companies, the product owner oversees the writing of user stories and arranges them on the product backlog. A *user story* is a very concise statement that only defines the user's end goal, whereas a *use case* frequently contains multiple extra phases. Use cases describe both a user's goal and the system's functional needs.

Six steps to writing a user story

- Decide what the end state will look like.
- Document tasks and subtasks.
- Determine your user personas:
  - Who is served in this story?
  - Which type of user or customer?
- Create stories as ordered steps.
- Seek user feedback.
- Draft stories that can be completed in one sprint.

In acceptance criteria, the product owner specifies what the story must accomplish in order to be accepted. In addition, they indicate when a user story is complete and working as intended.

Activity 2.1 **INDIVIDUAL ACTIVITY**

1. Define the term *problem solving*. (2)

2. List the FOUR categories of *calculational thinking*. (4)

3. Which of the following is the initial phase of the PDLC?

A Problem identification
B Testing
C Maintenance
D Design (1)

4. You are planning to develop a new software system for your organisation. You need to review the plans, models and architecture for how the software will be implemented. Of which of the following activities should you review the output?

A Requirements analysis
B Design
C Coding
D Testing (1)

5. You are planning to develop a new software system for your organisation. You need to verify that the implementation of the system matches with the requirements of the system. Which of the following activities would accomplish this requirement?

A Testing
B Design
C Release
D Requirements analysis (1)

Activity 2.1 (continued)

6. The product that you are developing is not yet finished, but you would like to release the product to a wider customer audience for feedback and testing. Under which of the following testing levels would this activity fall?
   - **A** Integration testing
   - **B** System testing
   - **C** Acceptance testing
   - **D** Regression testing (1)
7. True or false: A logic error will result in the program breaking. (1)
8. A … error is an error in the spelling and grammar of a programming language. (1)
9. Identify any THREE purposes of problem solving. (3)
10. List the FOUR steps of problem solving. (4)
11. Explain in your own understanding what Polya's first principle of understanding the problem entails. (2)

TOTAL: 21

2.2 Construct an algorithm and present a solution to a given problem

2.2.1 What is an *algorithm*?

An algorithm is a step-by-step description of how to arrive at a solution in the easiest way (refer to the definition provided earlier). Algorithms are not restricted to the world of computers; in fact, we use them in everyday life.

When addressing an issue with a programming language, the first step is to describe how to solve it! This will then lead the programmer to write an efficient code to complete the task. There are several ways to characterise an issue for which we want to develop a program. The method through which we explain a problem is known as an *algorithm*.

Characteristics of an algorithm

An algorithm must have the following characteristics:

- It should be straightforward and unambiguous. Each of its processes (or phases) and their inputs/outputs must be clear and lead to just one meaning.
- It should have zero or very well-defined inputs.
- It should have one or more well-defined outputs that match the desired output.
- Each phase must be distinct, well-defined and exact. There should be no room for doubt.
- It must end after a finite number of steps.
- An algorithm should have step-by-step instructions that are independent of any programming code.

2.2.2 Examples of algorithms in life

From tying our shoelaces to driving a car, everything we do involves a process that must be followed. An algorithm comprises a series of steps that help us to solve everyday problems as well as more complex ones in scientific fields. Your lecturer sorting papers in alphabetical order is an example of a sorting algorithm. A Google search is only possible with the help of algorithms. The next time you wait for a traffic light to change, consider the algorithm used to time the signals based on the flow of traffic.

Types of algorithms

Algorithms are classified based on the concepts they employ to accomplish a task. While there are numerous types of algorithms, Table 2.2 shows the most fundamental types of algorithms.

VOCABULARY

Recursive – *method of solving a calculational problem where the solution depends on solutions to smaller instances of the same problem*

ALGORITHM	EXPLANATION
Divide and conquer	Break the problem down into smaller sub-problems of the same type; solve those smaller problems; integrate those solutions to solve the original problem
Brute force	Try all possible solutions until a satisfactory solution is found
Backtracking	Solve problems recursively (see below) and discard solutions that do not meet the constraints of the problem
Greedy	Discover an optimum solution at the local level with the goal of discovering an optimal solution for the entire problem
Recursive	Solve the smallest and most basic version of a problem, then successively solve bigger variants of the problem until the original problem is solved

Table 2.2: Types of algorithms

The algorithm's performance may be measured using two criteria:

- Space complexity – the amount of space needed to solve a problem and generate an output
- Time complexity – the amount of time required to finish an algorithm's execution.

Drawing a kite

Let us take a kite as an example. Have you ever wondered where the idea of flying a kite came from? Kites are an ancient invention. They were used in China nearly 3 000 years ago. Kites were the first objects heavier than air to be flown by humans. The paper or fabric of which it is made stretches tightly between wooden or plastic strips to catch the wind. The kite is 'tied' to the ground by a string that is held by the person flying the kite and prevents it from flying off.

Let us look at the steps of drawing a kite.

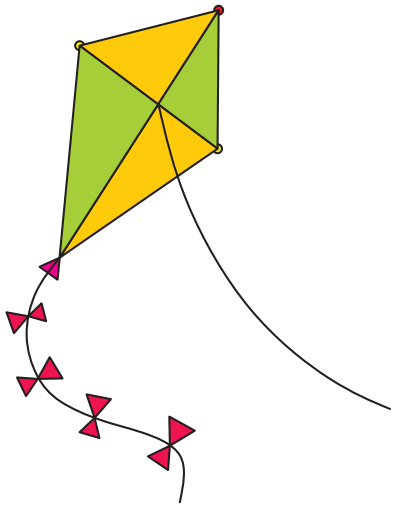


Figure 2.2: Image of a kite

EXAMPLE 2.1

Problem: Write an algorithm to draw a kite.

Solution

Step 1: Start

Step 2: To begin, draw a four-sided shape, such as a diagonally slanted diamond. The top two lines are noticeably shorter than the bottom two lines.

Step 3: Draw a pair of lines perpendicular to each other within the shape. Lines should be drawn from one corner of the shape to the other. These are the thin, lightweight wooden crossbars that hold up the kite.

Step 4: Draw a small circle in three of the shape's four corners. The bottom corner should have a small, curved triangle. The shapes represent how the kite is attached to the crossbars.

Step 5: From the bottom corner of the shape, draw a long, curving ling. This line is the string of the kite's tail.

Step 6: Along the string, draw two small triangles, connected to each other and to the string at one point. This forms the shape of a bow tied to the kite's tail.

Step 7: Draw another pair of triangles, touch points along the kite's string. This forms another bow.

Step 8: Draw a third set of triangles along the kite's tail, connected at the points to form a bow.

Step 9: Draw the fourth and final bow of the kite's tail, again using two small triangles connected by their points.

Step 10: Draw a long, curved line extending from the centre of the crossbars. This is the kite's string.

Step 11: Colour your kite. Kites are often brightly coloured so that they can be seen even high up in the sky.

Step 12: End

Let us look at another real-life example. Let's say we want to make lemon juice. Here are the steps:

EXAMPLE 2.2

Problem: Write an algorithm to make lemon juice.

Solution

Step 1: Start

Step 2: First, we will cut the lemon in half.

Step 3: Squeeze the lemon half as you press it onto the citrus juicer to extract its juice and collect it in a container.

Step 4: Add two tablespoons of sugar to the lemon juice.

Step 5: Stir the liquid until the sugar has dissolved.

Step 6: Once the sugar has dissolved, add some water and ice.

Step 7: Place the juice in the fridge for five to ten minutes.

EXAMPLE 2.2 (CONTINUED)

Step 8: Now it is ready to drink.
Step 9: End

Algorithms state instructions that must be followed in a particular order to accomplish a task. In the example of making lemon juice, specific steps must be performed to achieve the desired result.

2.2.3 Construct and devise an algorithm/basic instruction to complete similar tasks

Let us move on to use algorithms in solving calculational problems. We will start by listing the guidelines of developing the algorithms.

Guidelines for developing algorithms for programming situations

- An algorithm will be enclosed by `Start` (or `Begin`) and `Stop` (or `End`).
- To accept data from the user, generally used statements are `Input`, `Read`, `Get` or `Obtain`.
- To display a result or any message, generally used statements are `Print`, `Display` or `Write`.
- In general, `Compute` or `Calculate` are used to describe mathematical expressions and, depending on the context, applicable operators can be utilised.

EXAMPLE 2.3

Problem: Write an algorithm to find the average of six figures.

Solution

Step 1: Start
Step 2: Declare a variable sum with value 0.
Step 3: Get the sum of all the values in sum variable using a loop.
Step 4: Divide the sum by 6 and assign it to the average (avg) variable.
Step 5: Print the average.
Step 6: End

To learn how to write an algorithm, look at the following example.

EXAMPLE 2.4

Problem: Design an algorithm to add two numbers and display the result.

Solution

Step 1: Start
Step 2: Declare three integers: *a*, *b* and *c*.
Step 3: Define the value of *a* and *b*.

EXAMPLE 2.4 (CONTINUED)

Step 4: Add the values of *a* and *b*.
Step 5: Store the output of Step 4 to *c*.
Step 6: Print *c*.
Step 7: Stop

Here is another example of calculating all integers from 1 to 100.

EXAMPLE 2.5

Problem: Write an algorithm to calculate the sum of integers from 1 to 100.

Solution

Step 1: Start
Step 2: Initialise count $i = 1$, sum = 0.
Step 3: Sum = sum + i.
Step 4: Increment value of I; $i = i + 1$.
Step 5: Repeat steps 3 and 4 until $i > 100$.
Step 6: Display sum.
Step 7: Stop

Importance of understanding algorithms

You can use algorithmic thinking to break down issues and construct solutions in terms of discrete stages. Algorithms are applied in computers to sort and search elements.

Like any other design technique, algorithms have their advantages and disadvantages, as discussed below.

Advantages of using algorithms

- Independent of program languages
- Make program debugging easy.

Disadvantages of using algorithms

- Time-consuming for complex problems
- Understanding the logic in complex programs is very difficult.

2.2.4 Purpose of input, processing and output as part of the solution-creation process

VOCABULARY

Input – *what the user needs to punch into the computer through the keyboard for the program to start processing*

A computer program or other processes that employ the IPO pattern accept inputs from a user or other source (input), do calculations (process) on the inputs and return the results (output) of the calculations.

The system divides the work into three categories:
- Requirement from the environment (**input**)
- Calculation based on the requirement (process)
- Results or outcomes (output).

Processes are operations and activities that mediate the relationship between the input factors and the team's outputs. The computer does the processing based on how you have programmed the solution.

Outputs are the consequences (the result) of the process. This is displayed back to the user interface. Remember, if you enter the wrong input or use the wrong process, the output will also be incorrect – the principle of *garbage in, garbage out*. You cannot expect correct results if the input or process was incorrect.

2.2.5 Purpose of IPO chart in solution development

The IPO chart or model is a popular method for characterising the structure of an information-processing program or similar process in systems analysis and computer programming. An IPO table is the most fundamental framework for defining a process. It is introduced in many beginners' programming and systems analysis texts.

2.2.6 The various parts of an IPO chart

The IPO chart will consist of three columns representing input, process and output. Let us start with a simple mathematical challenge to accept two numbers from the keyboard and perform a mathematic calculation. The challenges are normally phrased as in the example below.

EXAMPLE 2.6

You are tasked by a client to develop a simple calculator that accepts numbers and can perform mathematic operations such as add, multiply, subtract and divide. The table outlines the layout of these elements.

MATHEMATIC OPERATIONS		
INPUT	**PROCESSING**	**OUTPUT**
• Number 1 • Number 2	• Assign variable for the number 1 • Assign variable for the number 2 • Select sign (+, –, /, *) • Calculate number 1 and number 2	Display the result of the calculation

Consider the following scenario:

You are required to write an IPO chart to convert Fahrenheit temperatures to Celsius. The formula is as follows:

$$°C = (°F - 32) \times \frac{5}{9}$$

The solution for the above scenario is presented in the example below.

EXAMPLE 2.7

MATHEMATIC OPERATIONS		
INPUT	**PROCESSING**	**OUTPUT**
Declare Celsius, Fahrenheit Input Fahrenheit	Calculate Celsius = (Fahrenheit –32) * $\frac{5}{9}$	Display Celsius

Task 2.1

Try to draw an IPO chart for converting Celsius to Fahrenheit.

Formula: °F = (°C × $\frac{9}{5}$) + 32

Show your answer to the lecturer.

Weaknesses of an IPO model

- The IPO model is very basic and fails to account for all the complicated relationships that occur in a computer program.
- Some of the 'processes' are not procedures at all, but rather team qualities that arise as the team works together. They are not actual events, but more like mediators of the input-output connection.
- The IPO model does not clearly represent repetitive tasks.

2.2.7 Flow charts

VOCABULARY

Flow chart – *diagram that shows the logic of the program or sequence of instructions in a single program*

A **flow chart** can be defined as a type of diagram that represents an algorithm, a workflow or a process. It shows the steps as boxes of various kinds. Arrows connect the boxes. This gives you a diagrammatic representation of the solution to a problem.

This method of conveying information was introduced in 1921 by Frank Gilbreth in a presentation called "Flow Process Chart". The process of drawing a flow chart for an algorithm is known as *flow charting*.

Some purposes of flow charts in solution development include the following:

- Programming – show a graphical representation of how a problem is solved.
- Workflow management and continuous improvement – represent the flow of data and processes and are used for improving systems
- Troubleshooting – to progressively narrow the range of possible solutions based on a series of criteria
- Regulatory and quality management requirements – business processes may be subject to regulatory requirements that require you to define and document your accounting procedures.

2.2.8 Outline the various symbols used as part of a flow chart

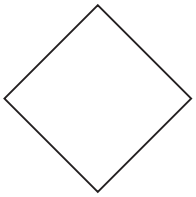
The different flow chart symbols have different conventional meanings.

- *Terminal symbol:* In the flow chart, we use a rectangle with rounded corners to represent the beginning and end of a program. The terminal symbol is shown below. This symbol must appear at the beginning and end of any program. **Some texts use an ellipse to represent terminal symbols.**

- *Input/Output symbol:* The input symbol represents the input data, whereas the output symbol represents the output process. The input/output sign is represented by the parallelogram shown below:

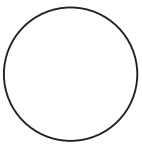
- *Processing symbol:* A rectangular shape is used to denote a process. Processes can be mathematic calculations or movement of data or instructions. The symbol given below is used to represent the processing symbol:

- *Decision symbol:* The diamond symbol is used to represent decisions. Below is the symbol for decisions. For simplicity, whenever there is a condition that needs to be met before certain statements are executed, you will need to add a decision symbol for that operation.



- *Flow lines:* These represent the exact sequence in which instructions are executed. Arrows are used to represent the flow lines in a flow chart. The symbol given below is used for representing the flow lines:

- *Connector symbol:* If flows are interrupted at one point and resumed on another page, the connector symbol is used. The connection symbol is shown below:



Guidelines for developing algorithms for programming situations

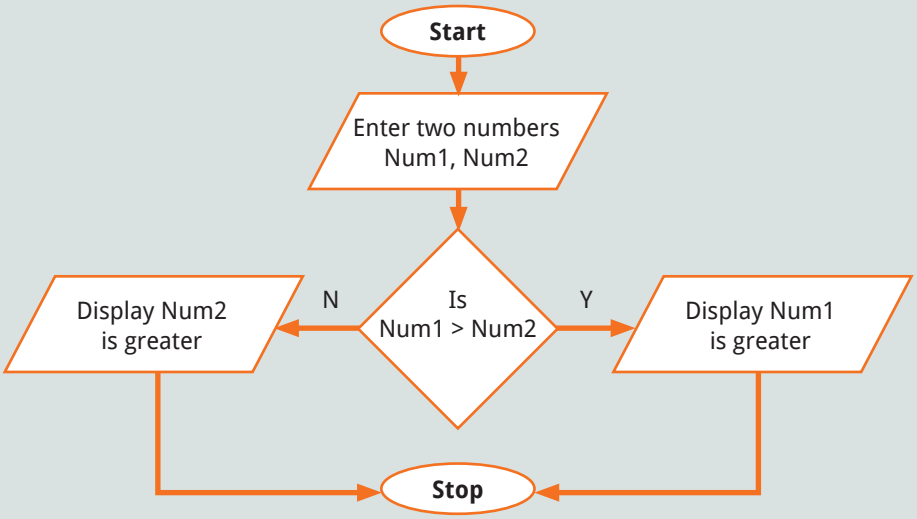
- Flow charts should have start and end points.
- Proper use of names and variables are needed.
- Only conventional flow chart symbols should be used.
- If the flow chart becomes large and complex, use connector symbols.
- Off-page connectors are referenced using letters.
- The general flow of processes is top to bottom or left to right.
- Arrows should not cross each other.

The next step is to use a flow chart to depict a given scenario.

EXAMPLE 2.8

Display the number that appears when a user enters two values from the keyboard using a flow chart. Assume that the two numbers are not equal.

Solution



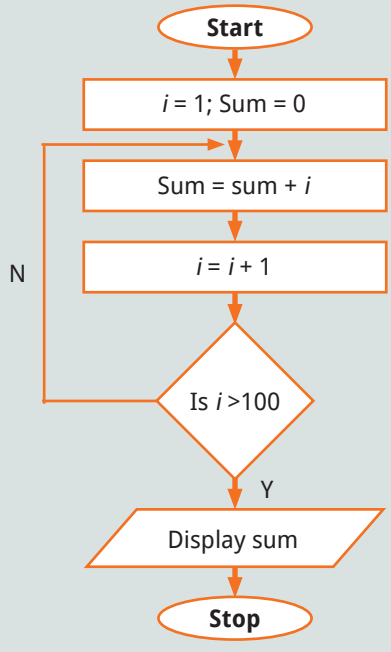
Comparing two numbers

Modify the solution above to cater for instances where the two numbers are equal. Show your answer to your lecturer.

EXAMPLE 2.9

Design a flow chart for a program that calculates the sum of integer numbers from 1 to 100.

Solution



Factorial of a number N

Advantages of using flow charts

- The logic of a program is communicated clearly.
- They can be used as working models when new software and programs are designed.
- They are used to assist in writing actual code in a high-level language.
- They make debugging a program easier.
- They can be used to analyse logical programs effectively.
- Testers benefit from flow charts.
- Using a flow chart makes maintaining the program easier.

Disadvantages of using flow charts

- They are time-consuming to create.
- It is difficult to create a flow chart for large, sophisticated software.
- Modifying an existing flow chart is problematic.

Some of the design techniques that are useful include decision tables and pseudocode. A **decision table** is a special kind of table that is divided into four parts by a pair of horizontal and vertical lines. A **pseudocode** is another tool used to describe the way to arrive at a solution. It is an informal, high-level description of an algorithm.

VOCABULARY

Decision table – *concise visual representation for specifying which actions to perform depending on given conditions*

Pseudocode – *detailed, readable description of what a computer program or algorithm must do, expressed in a natural language rather than a programming language*

2.2.9 Exploring and creating algorithms in the form of an IPO chart and a flow chart

There are three kinds of programming constructs: control structures, instructions and constructs. A program's instructions flow in sequence (one after the other), but there are instructions that allow code blocks to be repeated several times or let you choose which code to execute, namely:

- sequential constructs;
- selection/conditional structures; and
- repetition/loop/iteration structures.

Sequential constructs

These constructs simply describe one instruction followed by the next instruction. For instance, if a user is asked to enter any two numbers and calculate their sum, there is no choice or repetition to be made before solving the problem. The instructions must be followed sequentially. You just write what you have in mind line by line (related to programming).

Selection construct

A selection construct provides choices of statements to be executed by the program based on a given condition or conditions. For example, a program

can tell a user whether they are old enough to learn how to drive a car. If the user's age meets the required driving age, the program would follow one path and execute one set of statements. Otherwise, it would follow a different path and execute a different set of statements.

So-called *if statements* are typical examples of selection statements. For the scenario presented on the previous page, you can write your statements as follows in the form of a pseudocode:

```
OUTPUT "How old are you?"
age ← USER INPUT
IF age > 16 THEN
   Display "You are old enough to drive a car"
ELSE
   Display "Come back when you are older!"
END IF
```

You are going to learn more about `if` statements later. Consider the following example:

EXAMPLE 2.10

Problem:

Several employees work in a factory. If an employee is present, calculate the wage for eight hours; if an employee is absent, do not calculate the wage, just enter a zero (no work, no pay). The hourly rate = R250,00. So, depending on the state of the employee (whether present or not), you are asking the program to do one of two things:

a) When employee is present calculate wage; or

b) When employee is absent put wage as 0.

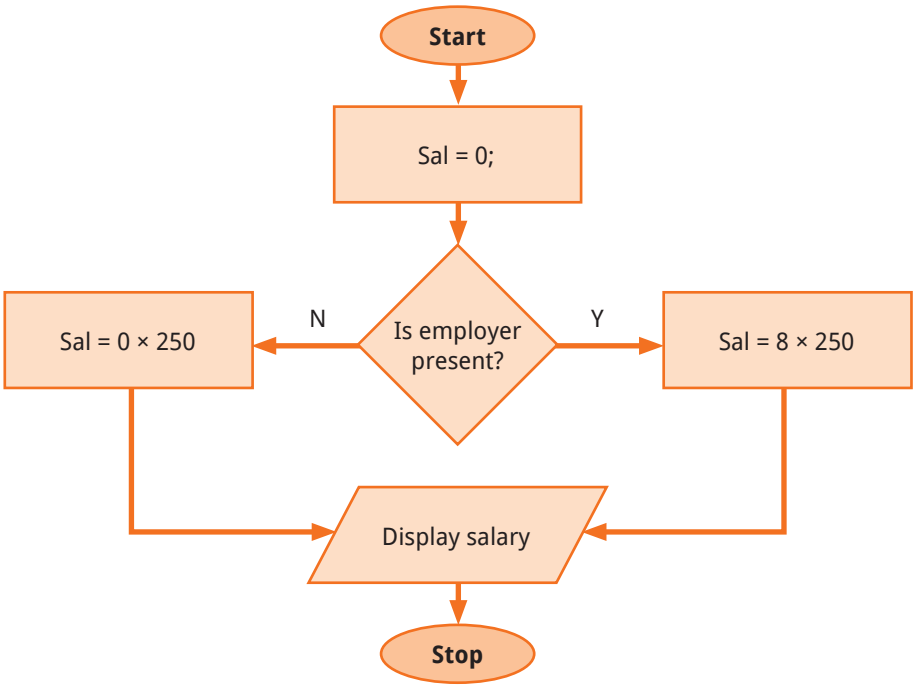
Solution

The processor will choose between taking "path a" or "path b", but it cannot do both.

This is an example of selection. Let us start by implementing an IPO table for the scenario.

INPUT	PROCESSING	OUTPUT
Initialise salary = 0	Check if employee is present If present hours = 8 Else hours = 0 Salary = hours × 250	Display salary

Flow chart depiction for the above scenario:



Flow chart depiction

In some instances, you can multiple decisions to be taken before a statement is executed. In this case you will be required to add multiple decision symbols.

Iteration/Repetition/Loop structure

Iteration occurs when a portion of code is repeated a certain number of times while a program is running or until a condition is satisfied. Because the program 'loops' back to earlier lines of code, the repeated process is also known as *iteration*. Instead of writing the same code again, programmers can write a section of code once and have the program run it repeatedly until it is no longer needed.

There are two types of iteration:

- Definite iteration (also known as *count-controlled iteration*)
- Indefinite iteration (also known as *condition-controlled iteration*).

When a program has to iterate a specific number of times, it is referred to as *definite iteration* and a FOR loop is used. A FOR loop employs an additional variable known as a *loop counter* that keeps track of the number of times the loop has been executed. Assume you want to print the following line of code five times:

```
OUTPUT  "Learning coding is fun"
OUTPUT  "Learning coding is fun"
OUTPUT  "Learning coding is fun"
OUTPUT  "Learning coding is fun"
OUTPUT  "Learning coding is fun"
```

The code could be written as:

For count in range (0, 5):

Print ("Learning coding is fun")

In the example on the previous page, *count* is our loop counter. The starting value and ending value are specified. You can also employ a *while loop* if you know the terminating condition. The actual syntax will be discussed in Module 3.

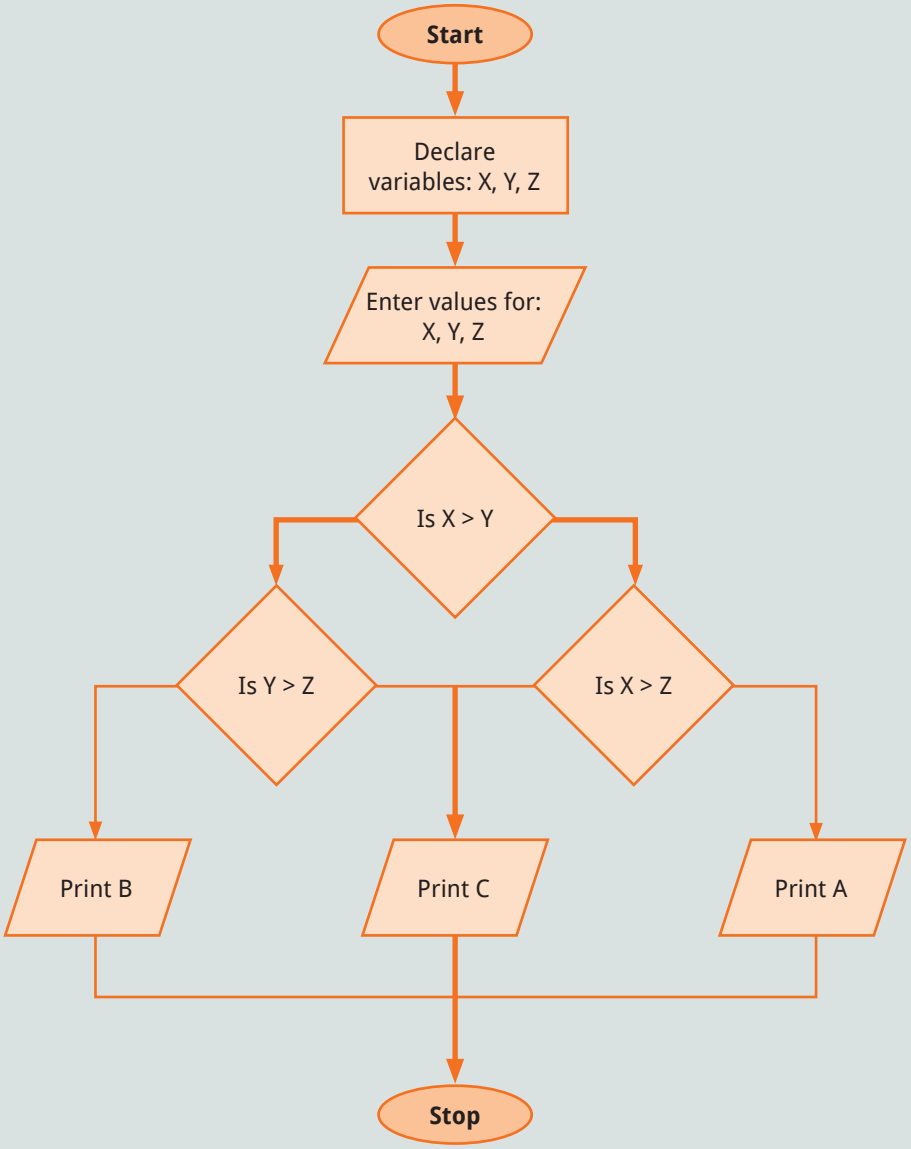
Here is a typical example of a loop structure.

EXAMPLE 2.11

Draw a flow chart to depict the following scenario. Enter any THREE unique numbers from the keyboard and display which one is greater.

INPUT	PROCESSING	OUTPUT
Enter three numbers: X, Y, Z	Compare if X > Y Compare if Y > Z Compare if X > Z	Display greatest value

Solution

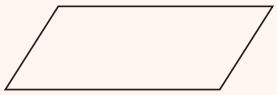
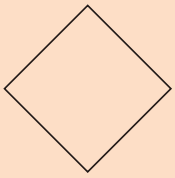
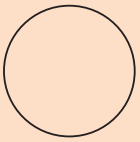


Flow chart for comparing three numbers

Activity 2.2 **INDIVIDUAL ACTIVITY**

1. . . . is a set of ordered and finite steps to solve a given problem. (1)
2. List any FOUR examples of algorithms. (4)
3. Write an algorithm to calculate the area of a rectangle. (6)
4. Write an algorithm to calculate the sum of integers from 1 to 100. (7)
5. The IPO chart will consists of three columns representing . . ., . . . and . . . . (3)
6. Draw an IPO chart to calculate the area of a rectangle whose sides are entered by the user through the keyboard. (5)
7. A . . . is a diagram that shows the logic of the program. (1)
8. Match the correct word with symbol as used in flow charts. Choose from the following list:

   - Process
   - Storage
   - Terminal
   - Connector
   - Flow
   - Input/Output
   - Decision

SYMBOL	REPRESENTS
	
	
	
	
	

(5)

9. Differentiate between a *selection construct* and an *iteration construct*. (4)
10. Which programming construct would you use to create a program for a multiplication table? (1)
11. Draw a flow chart to find the sum of all even numbers from 1 to N, where N is the number-terminating value. (11)

TOTAL: 48

Summative assessment

1. During which phase of PDLC is a software requirements document created? (1)

2. . . . take the entire source code and translate it into object code at once. (1)

3. One of the key phases in a PDLC is testing. List and explain FIVE different levels of software testing as applied in software development. (5)

4. List any THREE design techniques that are used during the design phase by systems analysts. (3)

5. Consider the scenario below:

R James Hardware has several bikes and tricycles for sale. There are 27 seats and 60 wheels altogether.

Determine how many bikes and how many tricycles there are. Apply Polya's steps to solve the problem. Outline your solution. (10)

6. List SIX characteristics of an algorithm. (6)

7. Write an algorithm to calculate the sum of the first five natural numbers. (7)

8. Discuss THREE disadvantages of using *flow charting*. (6)

9. What is another term for *repetition structure*? (1)

10. Draw a flow chart for computing the factorial of N, where $N! = 1 \times 2 \times 3 \times 4 \ldots N$. (8)

11. Draw a flow chart for the following scenario:

Customers get discount for buying products in a club's shop based on the membership type entered. There are three types of membership: gold, silver and bronze.

- If the membership is gold, the customer gets 25% discount and a message is displayed stating the discount given.
- If the membership is silver, the customer gets 20% discount and a message is displayed stating the discount given.
- If the membership is bronze, the customer gets 15% discount and a message is displayed stating the discount given.
- If the customer is not a member, 10% discount is given and a message is displayed stating the discount given. (17)

TOTAL: 65

Module 3

Concepts of programming for single-board microprocessors or microcontrollers

After you have completed this module, you should be able to:

- define the term *block-based/visual programming*;
- list examples of different block-based/visual programming languages and their uses;
- explore and use the visual programming language-development environment;
- construct/write using the visual tool, debug and run simple programs incorporating:
  - declaration of variables of different types
  - use and assign values to variables, incorporating program constructs with sequence structures;
- demonstrate knowledge of various concepts encapsulated in a block-based language;
- expose and apply various programming concepts as part of the coded solution;
- write code that applies programming language tools and constructs to draw various shapes (Turtle-type commands) on an output screen/window;
- devise a specific algorithm where applicable to solve a problem using user-defined code constructs;
- install Python on a single-board microprocessor;
- execute the commands to install Python 3 on the Raspberry Pi;
- execute the commands to install the Mu Python editor on the Raspberry Pi;
- start the Mu file editor and create a new Python file;
- add a simple `"Hello World"` print statement to an open Mu file;
- use the Mu editor to create a simple application;
- run/execute the source code;
- save the file;
- open an existing file;
- open and run/execute simple sample files;
- understand basic Python applications and the Turtle library;
- differentiate between a *compiler* and an *interpreter*;
- differentiate between a *shell* and an *IDE*;
- discuss the major characteristics of Python as an interpreted programming language;
- set up a simple Turtle program in Python;
- explain the purpose of the *from import* statement in Python:
- draw a line using the forward function;
- turn using the right and left functions;
- combine the draw, left and right functions to create simple drawings;
- explain how a colour is made up of a mix of red, green and blue;
- change the line colour using the colour function; and
- combine drawing with changing the colour.

Introduction

Visual programming is a programming language that allows users to illustrate processes. This enables program developers to explain the process in words that we can understand as opposed to languages that only the computer can process easily. The term *solution* refers to a set of related software programs and/or services. We start by looking at visual programming and solution development. Python is a high-level, interpreted, general-purpose programming language often used to build websites and software, automate tasks and conduct data analysis. Basic Python applications will be introduced in this module. You will also learn more about Turtle, a Python feature like a drawing board, which allows you to create pictures and shapes.

3.1 Visual programming and solution development

3.1.1 Defining *block-based* and *visual programming*

Block-based programming is an entry-level activity that uses a drag-and-drop approach to introduce beginners to computational thinking. It is a programming language that separates executable actions into modular sections called *blocks*. The blocks are usually represented by icons that you can click on and drag to reorder.

Visual programming allows us to describe processes through illustrations. Using a visual programming language (VPL) allows you to program using visual expressions, spatial arrangements of text and graphic symbols, either as a syntax element or as a secondary notation.

A VPL can also be classified into icon-based languages, form-based languages and diagram-based languages, depending on the type and extent of visual expression used. In VPL, graphic elements or icons can be manipulated interactively by users according to specific spatial grammars for program construction.

While a text-based programming language teaches the programmer to think like a computer, a visual programming language allows the programmer to describe the process in terms that we understand.

VOCABULARY

Block-based programming – *uses a drag-ang-drop learning environment where programmers use coding instruction blocks to construct animated stories and games*
Visual programming – *programming language that uses graphical elements and figures to develop a program*

3.1.2 Examples of block-based and visual programming languages

The benefits of learning programming are countless, from improving skills essential for an increasingly digital world to creative thinking. However, when it comes to learning the basic skills required, traditional text-based languages can be difficult to learn. Block-based and VPLs are far more user-friendly, even for non-expert users.

Examples of VPL include: Scratch, Bubble, Visual Logic, Apache Nifi, CiMPLE, Simulink, VIPLE and many others.

VPL are commonly used in:

- education;
- video games;
- multimedia;
- simulation automation; and
- business intelligence.

eLINK

~~For more~~ *examples of visual programming languages~~, visit the following link~~:*
futman.pub/Visualprogramming

Advantages of visual programming language

- It is easy to convert ideas into reality.
- Visuals are easy to understand.
- A VPL includes several built-in objects that are used during creation.

Disadvantages

- Because these languages use graphics, they require more memory; as a result, their execution is slow, and they consume a large amount of memory.
- Only operating systems that support graphics, such as Windows, Mac or Linux, can run these programs.
- There are only a few functions available in these languages.
- When the built-in functions are insufficient, you must add your own custom code.

3.1.3 Using the visual programming language development environment

Scratch will be our VPL in this course. Scratch is a coding platform that allows you to create games, stories and animations. You can create a wide range of projects, including Magic Pen, Wizard Tag Game, Geometry Dash, Basketball Game, Pacman and Snake. Scratch is one of the coding programs included with the Raspberry Pi. It can also be found online. To use the free online version, simply create a user and you are ready to go. The Scratch interface is depicted in the figure on the next page.

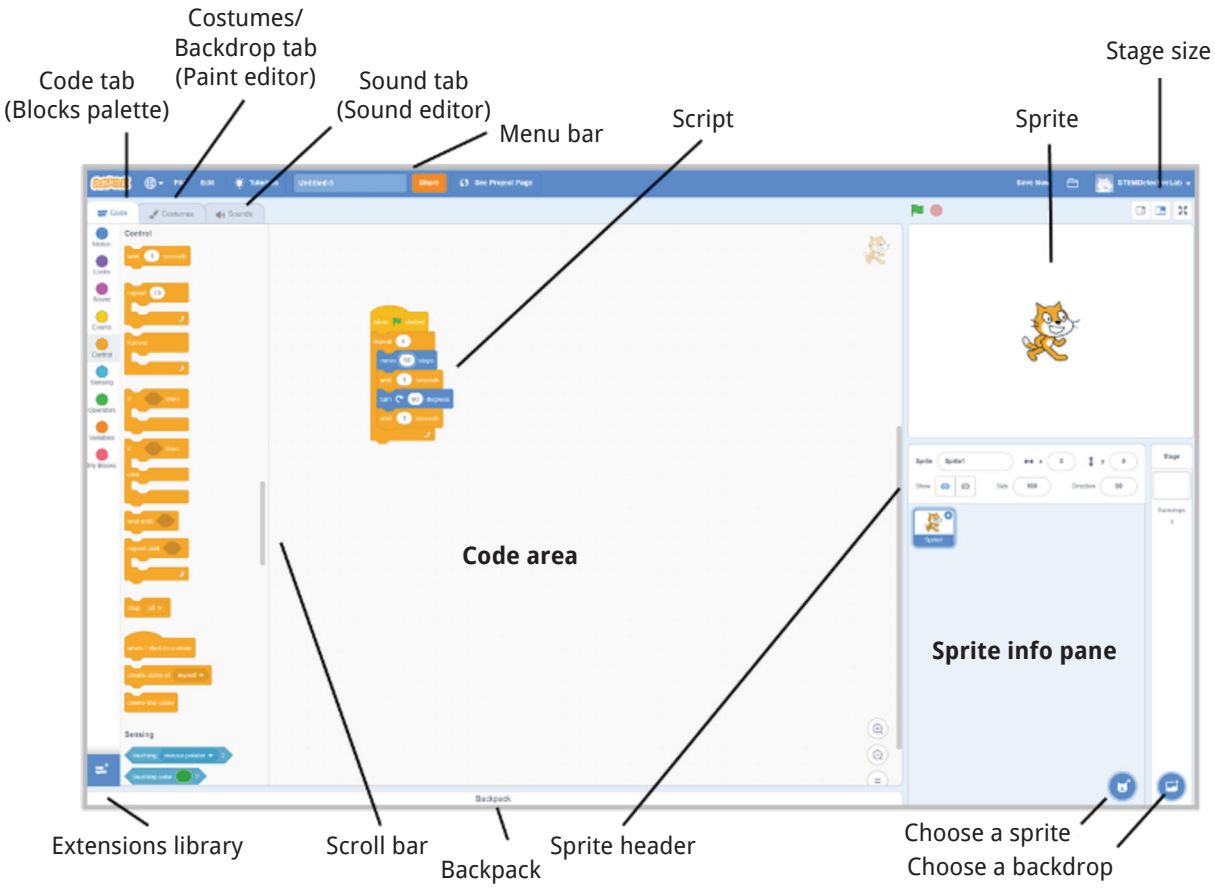


Figure 3.1: Scratch interface

Blocks palette

The *blocks palette* is the area that opens on the left of the screen when the `Code` tab is pressed. The left-hand side of the screen contains the nine block categories of Scratch. To the right of that, there is an area that contains a list of all the blocks. Scripts can be created by dragging these blocks into the *Code area*.

Extensions library

The *extensions library* is the blue button in the lower left-hand corner of the screen, below the Blocks palette. It allows the user to add extensions. One of the libraries we will use in this module is the *Raspberry Pi* and the *Pen* module.

VOCABULARY

Sprite – *two-dimensional image or animated image that is integrated into a larger environment and often plays a specific role*

Script – *program or sequence of instructions that are interpreted or carried out by another program, rather than by the computer processor*

Code area

The *Code area* is the large space to the right of the Blocks palette. It is an area for storing the blocks that run the project. Blocks can be dragged from the Blocks palette into the Code area and arranged to form scripts.

Sprite pane

The ***Sprite** pane* is located at the bottom of the stage, to the right of the **Script** area. It allows users to switch between sprites and view the different scripts each sprite contains.

Creating a new project in Scratch

Select the Raspberry Pi icon ➔ Programming ➔ Scratch3

This will open a Scratch window (like the one shown in the figure above) and the default Scratch sprite is the Cat. Sprites will be discussed in more detail later.

Creating a "Hello World" using Scratch and playing a specific sound

When you open the Scratch project window, you will notice that there are three tabs below the menu: Code, Costumes and Sounds.

Under *Code*, you will find the following subcategories (the Blocks menu): Motion, Looks, Sounds, Events, Control, Sensing, Operators, Variables and My Blocks.

Steps

Step 1: Start by selecting an event. Click Events (yellow) under the Code tab and select an *Event handler* to start your program.

Remember, you must drag the control to add it to the Script or Code area.

Figure 3.2: Event handler icon

DID YOU KNOW

An **event** *is an action or occurrence that can be identified by a program and has significance for system hardware or software.*

NOTE

To make the program work, you must click all the pieces together. Move the blocks closer until you see a white line, and then you can release the block. The blocks should now move as a unit.

Step 2: Click the *Looks* category/block (purple) under the Code tab and select a block that says *say Hello for( ) seconds*. You want to say "Hello World", so you have to click in the block and type in 'Hello World'. Change the number of seconds and type in '5' instead.

Step 3: To add sound, click the *Sound* option under the Code tab. You can select the sound of your choice by clicking the bottom right corner button, which will open a list of different sounds. Click the one you want to add to your program. Now drag the *Start sound* block and plug it below the *say Hello for( ) seconds* block in the code area. Finally, add the clear sound effects block to the bottom of the script. Our first program is ready to be tested.

Step 4: To run your program, click the green flag located just above the stage area where Cat (the sprite) moves. The text will be displayed for five seconds and then the meow sound will be played.

Note: The cat moves and changes direction following your input, as set out a little later.

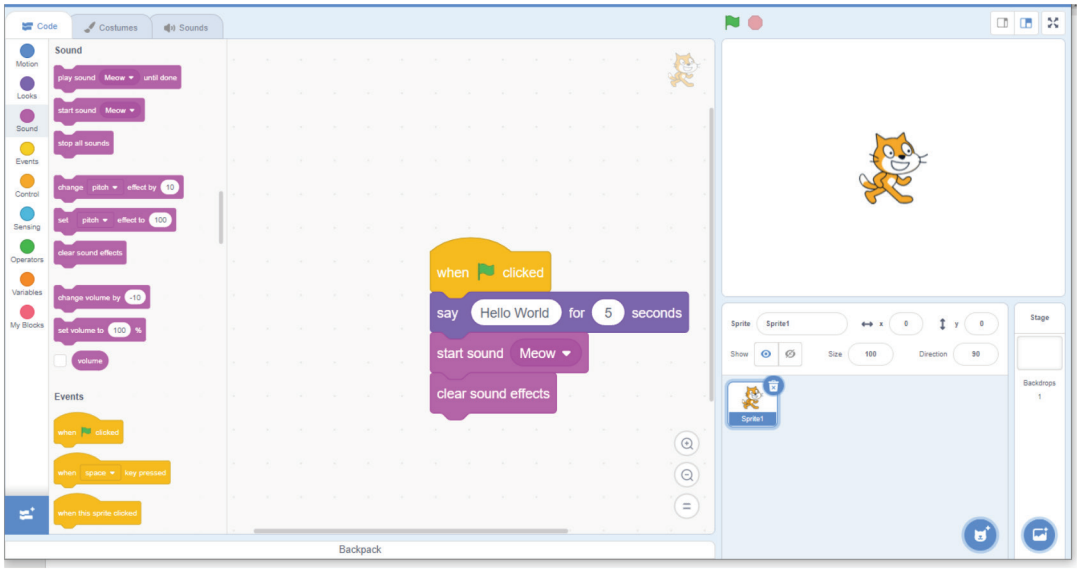


Figure 3.3: 'Hello World' and sound effect program

Saving a Scratch project onto your Raspberry Pi

Step 1: Select File ➔ Save to your computer projects.

Step 2: Type in the file name: *helloworld.sb3*.

NOTE

Scratch files are saved with an extension .sb3

How to open a Scratch project

Step 1: Click on the Raspberry Pi icon. Select Programming ➔ Scratch 3.

Step 2: Go to File ➔ Load on your computer and navigate to the path where your project is saved. You will find the file with a *.sb3* extension. In our case, select *helloworld.sb3*.

Step 3: Click Open.
You can now modify or rerun your program.

How to modify a saved Scratch project

In this example, we will modify the *helloworld.sb3* file.

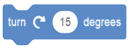
NOTE

To delete a block, simply drag it out of the code area.

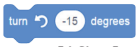
Step 1: Launch *helloworld.sb3*.

Step 2: Let us modify the project and code it to move the sprite 10 steps forward and 10 steps back. You will need to remove the *say Hello World for( ) seconds* block and replace it with the *move( ) steps* block under the *Motion* option.

Step 3: Click on *Motion* block in the programming palette. Edit the value 10 and make it 100.

Step 4: Go back to the Motion block, select the 

option and add it to the script.

Step 5: Go back to the Motion block, select the turn 

and add it to the script. Take note that the value has been modified to –15.

Step 6: Click on Motion block in the programming palette again; now click to 

move the sprite 10 steps. Edit the value 10 and make it –100.

Step 7: Our script is complete and will appear as shown in Figure 3.4.
Test the script by clicking on the green flag. You are going to see a clockwise movement each time the green flag is clicked.

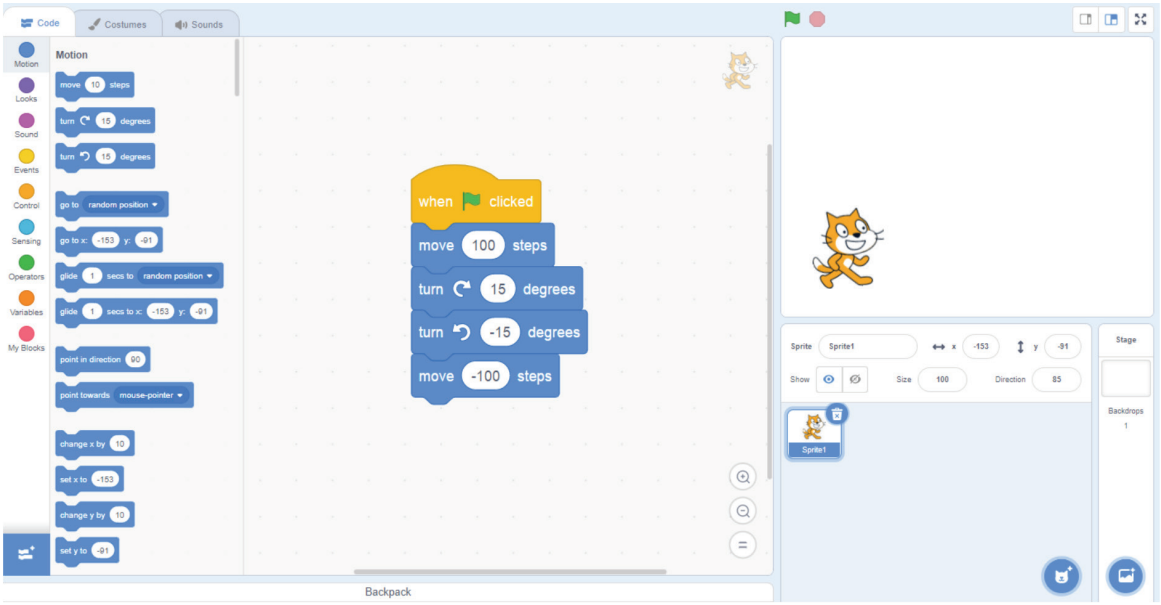


Figure 3.4: The modified project

Try to add different blocks from each of the palettes and see the effects applied to the sprite.

Looping

So far, the instructions have been executed in sequence or *sequentially*. How would you go about repeating an event several times? Scratch has a control block known as a *loop*. Repetitive execution of the same block of code over and over is referred to as *looping* or *iteration*. Let us create a program where the movement of the sprite has a looping control that will repeat the movements of the previous example, but run continuously.

EXAMPLE 3.1

Step 1: From the Events palette, select the green flag block.

Step 2: Click Control in the Blocks palette and select the *forever* block.

Step 3: Change the *move( )* steps block to move the sprite 100 steps and click on Sound to insert the 'Meow' sound you want to repeat.

Step 4: Add a *wait seconds* block and slot it in-between the *play sound until done* and the *move – 100 steps* blocks.

Step 5: Now click on the green flag to run the program.

The program will run forever, with the sprite producing the meow sound until you click the small octagon shape next to the green flag to terminate the process. The figure alongside illustrates the looping code.

Looping structure

3.1.4 Program constructs and sequence structures

VOCABULARY

Variable – *value that can vary, in other words, change over time and under the control of the program*

In programming, **variables** are labels for locations in memory used to store values. The value of a variable can be a number, text, true or false or empty (known as a null value).

Variables are powerful tools. Think of the things that you have to track in a game: the health of a character, the speed of a moving object, the level currently being played and the score. All of these are tracked as variables. In Scratch, you can create variables in two different ways, namely by using either a:

- built-in variable; or
- user-defined variable.

A *built-in* variable is one already found under the variables category. By default, my variable is defined. A *user-defined variable* is declared in case the user wants to create his or her own variable.

How to create a variable

In this section, you will learn how to create a variable and use it to do a simple addition of two numbers.

To add two numbers and display the answer, you will need three variables:

- `Num1:` to store the value of the first number
- `Num2:` to store the value of the second answer
- `Sum:` to store the result

```
Sum = num1 + num2;
```

Start Scratch and name your program *variables.sb3*.

Step 1: Click on Variables in the Code tab and select Make a Variable. Type in the name of your variable, which is `num1`, and click OK. Select Make a Variable again and type in the second variable, `num2`. Click OK. Finally, you will need a sum, so add a third variable, type in `sum` and click OK.

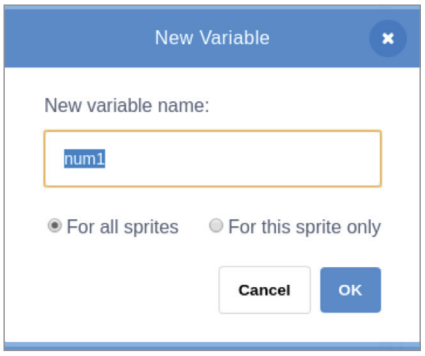


Figure 3.5: Creating a variable

Step 2: Click on Events and select the event handler (yellow with a green flag).

Step 3: Click Sensing (light blue) under the Code tab and drag the *ask( ) and wait* block to the coding area.
Delete the standard text `"What's your name?"` and replace it with `'Enter firstnumber'`.

Step 4: Go to Variables (orange) and select the *set* block. By default, it will set the first variable `num1` to `0`. You need to change this by selecting the *set( ) to answer* (from the Sensing block palette) and type in `num1`.

Step 5: Repeat Step 3 and change the text to `"Enter second number"`.

Step 6: Repeat Step 4 and *set num2 to answer*.

Step 7: Add another *set* block. The `sum` variable will add the two numbers together using an *Operators* block (green). Drag the + operator and add `num1 + num2` to the *set sum to* block.

Step 8: Click on Looks (purple) in the Code tab. Drag the *say sum for( ) seconds* block to the code area. The answer will remain on the screen for the number of seconds entered.

Step 9: Run your program and enter the values of `num1` and `num2`. The sprite will display your answer.

Solution in the code area:

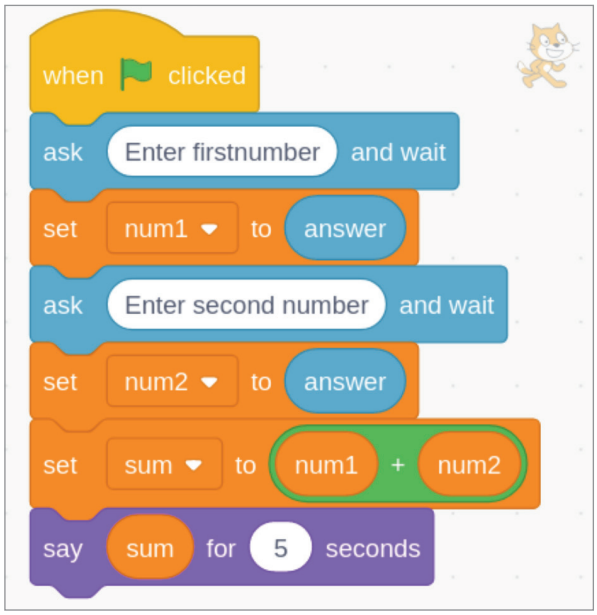


Figure 3.6: Adding two numbers

DID YOU KNOW

You can change the sprite to suit the category you want by navigating to the bottom right corner of the sprite area and clicking the icon of your choice. To delete the cat (default sprite), simply click on it and select another sprite – the ways in which this can be done will be explained later. You can also set the background by clicking the icon at the bottom corner of the stage area.

3.1.5 Concepts encapsulated in a block-based language

Encapsulation refers to the bundling of data, along with the methods that operate on that data, into a single unit. Encapsulation may also refer to a mechanism of restricting the direct access to some components of an object, such that users cannot access the state values for all the variables of a particular object. In this section, you will learn more about some of the constructs used in block-based programming languages.

Differentiating between a sprite and an image

The term *sprite* was briefly defined earlier. It is a type of stand-alone, 2D computer graphic character that you can control using code. The sprite plays a specific role. It can often be independently manipulated within a larger image environment. Each sprite has its own script, costume and sound. It can move independently. Sprites appear in the Sprites pane below the stage in the interface.

The term *computer image* may refer to computer-generated imagery, which is still or moving imagery created by or with the help of a computer. In Scratch, the image can be drawn by the user using the Backdrop option or it can be uploaded into Scratch from external sources. The images can be converted to bitmaps or vectors and the user can then apply them in their code.

Using existing images and sprites

The existing computer images are the pictures or photos available on your computer. These include stock photos, vectors and illustrations that are available for free. To use existing images, the user must click on the Backdrop tab and select *upload backdrop* from the options available when you point the cursor at the *Choose a backdrop icon* at the bottom corner within the backdrop section.

The sprite feature is one of the fun elements of Scratch. As mentioned previously, you can easily change the sprite, either by choosing one from those provided or by creating your own entirely.

In the Sprite pane, when you point to the choose sprite icon  , you will be presented with the following four options for the sprite:

- The Search button allows one to choose a sprite from the library. There is a wide variety of sprites that are prebuilt into Scratch. They are grouped into different categories. You can choose any of the existing sprites by clicking on the icon *Choose a Sprite* pane. It will show up in your sprite list.
- The Paintbrush button creates a blank sprite with an empty costume. You can use the paint editor to create your own sprite and the default name will be *sprite2*.
- The Surprise button creates a random sprite.
- The Upload button allows you to upload an image that you may have downloaded from the Internet or drawn yourself. Simply choose the image that you want to add as a sprite.

User-defined sprites

As mentioned above, you can create your sprite by simply clicking on the paintbrush, using the surprise button or uploading an image from your computer and converting it to Bitmap. The default name for your sprite will be *sprite2*.

Categories of block-based coding blocks

Scratch's VPL does not use commands. It uses coding blocks instead. Programs are created by arranging (stacking) the blocks in the appropriate order.

The following four types of blocks are used in Scratch:

- Command blocks
- Blocks that link events with the running of the program
- Blocks that control the running of the program
- Blocks that report a certain value (function blocks).

Command blocks (usually blue) have notches at the top and bumps on the bottom. Their shape allows them to be connected to other blocks.

Figure 3.7: Command blocks

NOTE

These blocks act like triggers – they start the running of scripts attached to them. The example above means that any command block attached to it will only run when the green flag located above the stage is clicked.

The blocks that link events with the running of the program are usually yellow and have 'hats'. They can therefore only be placed at the top of the stack of blocks. The bump at the bottom allows new blocks to be added to them.

Figure 3.8: Blocks that link events

Blocks that control the running of the program (usually orange) can also accommodate other blocks above or below them. What is more, blocks can also be put *inside* them, as illustrated below.

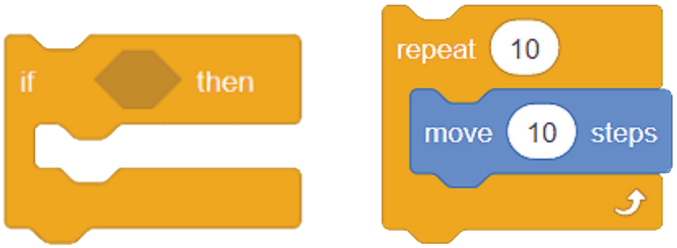


Figure 3.9: Blocks that control the running of the program

Blocks that report a certain value (function blocks) differ from other blocks in that, apart from their appearance, they cannot be integrated into a script on their own. They cannot be added to other blocks; instead, they become part of the block into which they are inserted.

Figure 3.10: Function blocks

Function blocks contain certain values. Some have rounded edges, while others are angular. They are inserted into input fields in command blocks and blocks that control the running of the program.

Scratch's Blocks palette sorts blocks by categories. Each block within a category has the same colour. The separate colours used within categories make it easier to distinguish parts of a project and the user can also access the blocks more easily. In Scratch 3.0, there are nine block categories, eleven extensions and three Raspberry Pi extensions.

The following nine blocks are used:

- Motion – deal with the movement of a sprite
- Looks – related to the appearance of sprites and the stage
- Sound – add sounds to the app
- Events – related to various triggers in a project
- Control – run the basic flow of a project in the desired fashion, i.e. they control scripts; also provide functions for looping various blocks and scripts
- Sensing – associated with sprites and the stage to detect conditions
- Operators – perform arithmetic functions within a project
- Variables – used for storing information, such as variables and lists
- My Blocks (originally called More Blocks) – user-made custom blocks that hold procedures for a selected sprite.

The following three extensions are available with the Raspberry Pi version of Scratch:

- Raspberry Pi GPIO extension – allows Scratch to control Raspberry Pi GPIO (general-purpose input/output) pins
- Raspberry Pi Sense HAT extension – allows Scratch to interact with the Raspberry Pi Sense HAT, a grid of lights that can be used to display messages
- Raspberry Pi Simple Electronics extension – allows Scratch to control lights connected to a Raspberry Pi.

Explore the Raspberry Pi

With the Raspberry Pi 4, you can now program in the latest version of Scratch. In addition, you can use the new Scratch Raspberry Pi extensions to interact with Scratch in a more physical way. When people think of programming or coding, they usually think of software. Coding can be about more than just software, although it can affect the real world through hardware. Physical computing is all about controlling things in the real world with programs: setting the program on your washing machine, changing the temperature on your programmable thermostat or pressing a button at traffic lights to cross the road safely. We are going to introduce the physical component that interacts with your programs on the Raspberry Pi: the GPIO header.

GPIO

GPIO stands for general-purpose input/output. One of the powerful features of the Raspberry Pi is the row of GPIO pins along the top edge of the board. These pins are a physical interface between the Raspberry Pi and the outside world. At the simplest level, you can think of them as switches that you can turn on or off (input) or that the Pi can turn on or off (output). There are 40 pins on the Raspberry Pi (26 pins on earlier models), and they provide various functions.

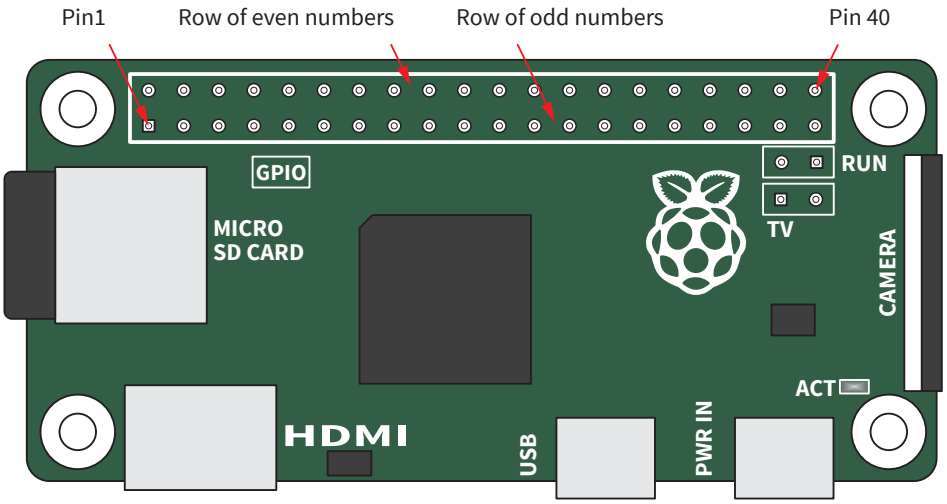


Figure 3.11: GPIO header

The Raspberry Pi pins are numbered, as shown in the figure above. The inner row consists of uneven-number pins from pin 1 to pin 39. The outer row consists of even-number pins, i.e. pin 2 to pin 40.

NOTE

If you follow the instructions, playing with the GPIO pins is safe and can be fun. Randomly plugging wires and power sources into your Pi, however, may destroy it, especially when using the 5-V pins.

DEMONSTRATION: USING SCRATCH TO PROGRAM THE PI TO LIGHT UP AN LED

You will need:

- 1 × LED (any colour)
- 1 × 220- or 330-Ω resistor
- 2 × female-to-male jumper wires
- 1 × breadboard

Instructions

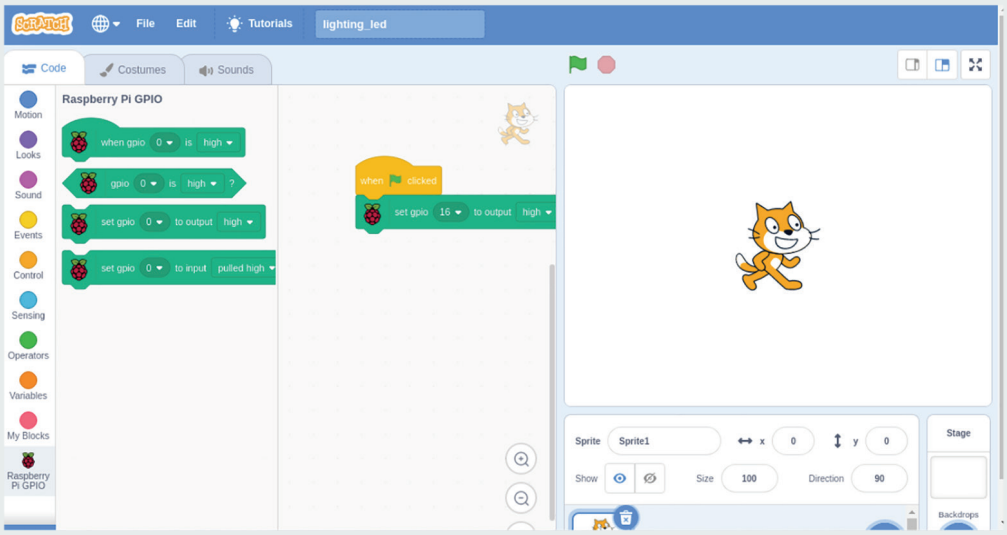
1. Open the GPIO header (usually covered with rubber to protect the pins).
2. Connect the Raspberry Pi to the ground rail of the breadboard, taking care to align the pins – no pins should be bent or crushed.
3. Connect the pins from the Raspberry Pi to the breadboard using jumper cables.
4. Connect the LED to the breadboard. An LED has two legs – the longer one is the positive and the shorter one the negative lead. Insert the LED with the two component legs on separate rows.
5. Connect the negative lead of the LED to the ground rail using a jumper wire.

NOTE

Breadboards come in different sizes. For the purposes of this module, a small breadboard with 30 small holes in each column will be used. Breadboards of different sizes would have been used in the Robotics course.

DEMONSTRATION (CONTINUED)

6. Connect the LED to Pin 28 via the resistor. One of the legs of the resistor must be in line with the positive lead on the LED. Slot the other leg into any unused breadboard row.
7. Connect the negative leg of the LED in the same row as the negative leg of the LED.
8. Connect the positive jumper cable of the breadboard to GPIO16 (which is Pin 36).
9. Connect the negative jumper cable of the breadboard to the ground pin. In this example, I used Pin 39.
10. We need to add a few blocks to the toolkit to access our GPIO pins and turn on the LED. In the middle panel, click More Blocks. Now click Add an Extension and choose *Pi GPIO*. This will add blocks you can use with the Raspberry pins.
11. Add a green flag from the Events block.
12. Add the *set gpio( ) to output high* block. Set the GPIO to 16. Click the green flag to run the Scratch program. You will see the light turns on. Congratulations! We have set out the first project where hardware is interacting with software. See the simple script in the figure below.



In the above example, a forever block was added, but the sequential option was used in our program. Loops will be implemented in the subsequent program. It is important to understand the basics.

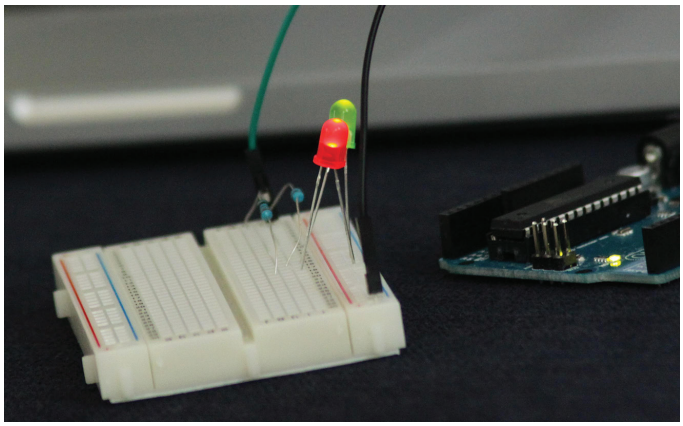


Figure 3.12: Blinking LED light

To modify the program above to have a blinking LED light, follow the steps set out here.

Step 1: Drag the green flag block from the *Events* option to the code area.

Step 2: Add the *wait( ) seconds* block. Type in 1 second.

Step 3: Add the *set gpio( ) to output high block*. Set the GPIO to 16 and change the output to *low*.

Step 4: Add another *wait for( ) seconds* block.

Step 5: Add a *forever* block and slot in the blocks you have created, as indicated in Figure 3.13.

Step 6: Click on the green flag and you will see that the LED is blinking.

Figure 3.13: Code for a blinking LED

NOTE

You will have to use the Repeat *block in the* Control *options.*

Changing appearance and costumes

A *costume* is one of the many 'frames' or alternate appearances of a sprite. Sprites can change their look to any of their costumes. There are two different costume types:

- Bitmap, which uses pixels to create an image
- Vector, which uses two points to create a line.

You can change a sprite's appearance by modifying its costume.

Adding buttons

Scratch allows the user to create custom buttons by clicking on Choose sprite and searching for buttons. Click on the shape you like. To customise the button, go to Costumes and add text to indicate what the button does. Clicking outside the button allows you to change the size of the text or reposition it.

eLINK

To learn more about creating custom buttons, visit the following link:
https://www.youtube.com/embed/DZX7Z6LCl5A?start=11

Assigning events and triggers

In section 3.1.3, you learned how to create a new Scratch project. Step 1 is to click on the *Events* block. As mentioned previously, event blocks are used to sense events, which trigger the attached code to run. They are therefore

essential for every project. You have already learned how to add an event to the code by simply selecting different event blocks. For example, in the LED project, we started by adding the green flag block. That is an event block that starts the project, i.e. it acts like a trigger.

Creating and using variables

The concept of variables was explained in section 3.1.4, where a variable was created to add two numbers. Variables are simply containers of data. They have a name and a value.

Operations on data

Operators

Operators are the active elements of a computer program. They perform actions such as adding two variables, dividing one variable by another variable, comparing one variable to another variable, etc.

Operands

According to the current jargon, *operators* operate on *operands*. For example, in the following expression, the plus character is an operator, while `number_1` and `number_2` are operands.

```
number_1 + number_2
```

If `number_1` and `number_2` are numeric variables, this expression produces the sum of the values stored in the variables named `number_1` and `number_2`. The variable *x* would be called the *left operand* and the variable *y* would be called the *right operand*. Computer programs in many languages consist of statements that, in turn, consist of expressions.

Comparisons

In scratch, you can compare two or more variables or values using three operators. The values are compared to determine whether the statement is true or false. If the result of the comparison is true, the following code will be executed. If the comparison is false, the second block under else will be executed. When the green flag is clicked, the program will prompt the user to enter a number. The number is then compared to 5.

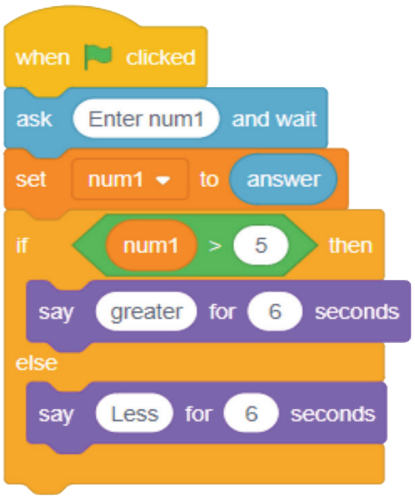


Figure 3.14: Comparison of logic

Broadcasting

A *broadcast* block is both an Events block and a stack block. Scratch uses this signal to send or broadcast messages throughout the program. Any scripts in any sprites that are hatted with the *when I receive*( ) block set to a specified broadcast will activate. The broadcast block allows scripts to send broadcasts without delays.

Repeating actions

The *repeat( )* block is a Control block. Blocks held inside the repeat block will loop a given number of times, before allowing the script to stop or continue. In section 3.1.3 you learned how to create a loop. The repeating action was also used to blink the LED.

Using conditional statements

A conditional block executes specific statements based on a given condition. The *if( ) then* block or *if then*( ) *else* block is a control block. The use of conditionals is demonstrated in the section on Comparisons on the previous page. A set of lines will be executed if a condition is true. If the condition is false, the *else* part is executed.

VOCABULARY

String – *data values that are made up of ordered sequences of characters, such as* `"Hello World"`

List – *number of items in an ordered or unordered structure*

Manipulating strings

A **string** is a sequence of characters in a computer, including spaces. Although in some circumstances they are limited to 10 240 characters, any character may be used and can appear in certain blocks, mainly in the Operators section.

Basic list operations

A **list** is a tool that can be used to store information. List blocks are a subcategory of the Variables blocks group. They work like a variable that holds multiple variables inside it. They are colour-coded dark red and are used to manipulate lists. Scratch 3.0 has the following seven list stack blocks:

Figure 3.15: List stack blocks

Incorporating sounds and images

In section 3.1.3, we looked at how sound can be added to a project. Some sprites come with their own sound. Select the sprite and then click on the *Sounds* menu above the Blocks palette. Sounds associated with your sprite

will appear on the left of the screen. To select a different sound, go to *choose a sound* in the bottom left corner of the *Sounds* menu and select the magnifying glass icon. This will take you to the Scratch sound library where you will find various sound options to choose from. Add the sound you like by left-clicking on it. The sound will appear in the left-hand menu under the Sounds tab. You can also create your own sound opening the choose your sounds menu and clicking on the microphone icon. Record the sound and save it.

Refer to section 3.1.5 where incorporating images is discussed.

Incorporating multiple sprites

REMEMBER

Each sprite has its own script area. When you click on the sprite, the program will appear.

Scratch allows the programmer to use two sprites in one program. This is quite handy when you are creating games. In the Sprites area at the bottom of the screen you will find the *New Spite* option. You can:

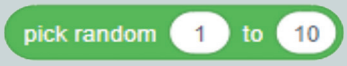
- choose a sprite from the *Sprite Library* by clicking the sprite icon;
- draw your own sprite by clicking the paintbrush icon;
- load a picture by clicking on the file icon; or
- take a picture by clicking on the camera icon.

We are going to create a game to illustrate how multiple sprites can be incorporated into a Scratch program.

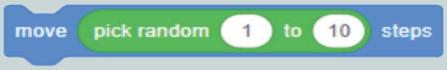
BEETLE BUTTERFLY GAME

- Remove the cat sprite and choose a butterfly (e.g. Butterfly 1) from the Category section in the Sprite Library.
- Next select a Beetle.
- Now add a Backdrop or working environment to the stage. Click the Backdrop button (it looks like a little mountain with a sun) to open the *Backdrops* library and choose the backdrop.
- Select the Butterfly and add an Events block – 

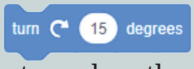
- Click Control and select a *forever* block.
- Click on Motion in the Blocks palette. Select the *move*( ) *steps* block and insert the value 15.
- Navigate to Operators and add the *pick random*( ) to ( ) block:



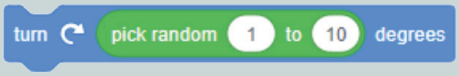
- The ultimate block will look as follows:



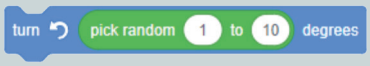
Now slot this block into the *forever* block.

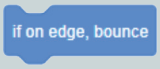
- Add the 

block and plug the *pick random* block into the circle to replace the number 15. The new block will look as follows:



- Add another *turn( ) degrees* block and repeat the previous step so that the resulting block looks as follows:



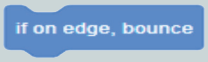
- Now add the *if on edge, bounce* block 

If you click on the green flag, you can see the butterfly moving in different directions. When it reaches the edge, it bounces back as programmed.

Next, we need to program the second sprite which is the Beetle. The idea of the game is that when the sprites (Beetle and Butterfly) collide, the Beetle must change its colour.

To program the second sprite, select the sprite, by simply clicking on the Beetle and follow the steps set out here:

- Select the Butterfly and add an Events block – 

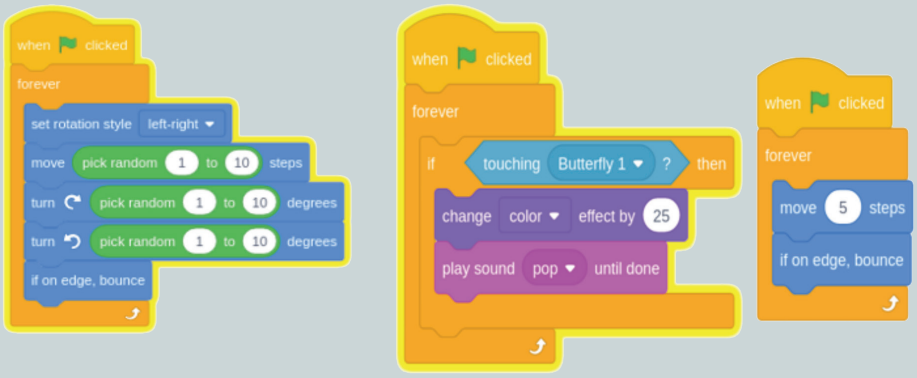
the green flag event.
- Click Control and add a select *forever* block.
- Add the *move (5) steps*.
- Next we want to add the *if on edge, bounce* block 

- Now we want to add another script for the Beetle in the same code area.
- Select the Butterfly and add an Events block – 

- Add a *forever* block.
- Add the *if( ) then* block and add a Sensing block *touching( )*? Change from mouse-pointer to Butterfly:

- From the *Looks* block select the *change color byeffect by 25*.
- Add a sound of your choice.
- Now hit the flag button and once the Butterfly touches the Beetle, it will change colour.

The code should look like the one below. The first screenshot is for the Butterfly and the last two are for the Beetle.



Code for the game

3.1.6 Various programming concepts as part of the coded solution

In this section we are going to look at incorporate variables.

The use of variables

The use of *variables* (a changeable value recorded in Scratch's memory) was explained in detail in section 3.1.4. Unlike lists, variables can only hold one value at a time. These values can be either numbers or strings – any text. Clicking on an isolated variable in the scripts area displays a small bubble reporting the value of the variable. Unlike many other programming languages, Scratch does not allow variables to be created by a script as it runs. Instead, variables are created with the Make a Variable button in the block palette.

Variable naming conventions

Variable names must be meaningful and must be related to what you are doing. When naming variables, avoid starting with capital letters. No spaces should appear in variable names.

There are three types of variables in Scratch:

- global
- local
- cloud variables.

Global variables

A *global variable* is a default variable. It can be read and changed by any sprite or the stage, regardless of which sprite it was created on. All variables are stored in RAM and are default to the values in the file from which they originate. The user is presented with the following options: *Choose for all sprites* or *Choose and for this sprite only?* If the former is chosen, then that variable becomes global and can be accessed by anyone who needs it.

Local variables

Local (private/personal) variables are created in the same way as global ones, but the user selects the *Choose and for this sprite only* option. The variable can only be changed or accessed from the sprite on which it was created. No other sprite can access it.

Cloud variables

Cloud variables allow users to store variables on the server of the scratch. They are indicated by a cloud-like symbol and can update themselves.

Assigning values to variables

This category of operators is used to assign values to a variable. The = operator is an assignment operator, but it is not the only one. Scratch 3.0 has the following operators: seven **Boolean** and 11 Operator Reporter blocks. Select the *Operators* block and you will see each of them.

Data types in Scratch

Scratch has built-in support for three data types that you can use in blocks: Booleans, numbers and strings.

- Boolean – can have only one of two values: true or false
- Numbers – this variable can hold both integers and decimal values

VOCABULARY

Boolean – *system of algebraic notation used to represent logical propositions by means of the binary digits 0 (falses) and 1 (true)*

- String – a sequence of characters that can include letters (both upper- and lowercase), numbers (0 to 9) and other symbols that you can type on your keyboard (+, –, &, @, etc.).

Input and output of information, messages and values

The user's input is saved in the answer function block. The output can be in form of a message attached to the sprite.

Operators and precedence

Scratch allows a combination of different operators. *Order of Operations* is one concept in the traditional mathematics curriculum that programmers quickly forget. Look at the green Operators menu in Scratch.

Note that the four Arithmetic Operators, the random number picker, string operators (join, letter, length), modulus, round and square root operators are rounded at each end. The relational (<, =, >) and Boolean operators (and, or, not) are pointed at each end. The rounded ends represent parenthesis.

Figure 3.16: Operator precedence

If you look carefully, you will see that the parenthesis (2 * 3) tells us that the block will evaluate as 7. The order can be remembered by using the acronym PEMDAS – parentheses, exponents multiplication, division, addition and subtraction.

Comparison operators and performing logical comparisons

In the Beetle Butterfly game, we made a logical comparison. There are three logic operators in Scratch:

- and
- or
- not.

OPERATOR	MEANING
	The result is true only if the two expressions are true.
	The result is true if either of the two expressions is true.
	The result is true if the expression is false.

Table 3:1: Example of logical operators

Let us say the program checks whether someone's age is greater than 65 but below 100. It should display a message to say "You are now supposed to go and rest" or "You must work for more years".

The code would look like the one below:

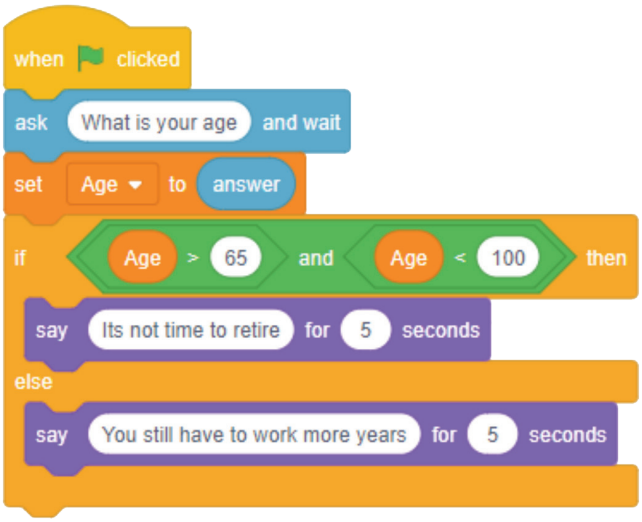


Figure 3.17: Comparisons

Calculations such as area and volume

Arithmetic operations are one of the operations performed by programming languages. In this section, we will look at a few such calculations.

Calculating the area of a triangle

The code in the figure below calculates the area of a triangle. The user must enter the height and base. You can test your program by following the code provided here.

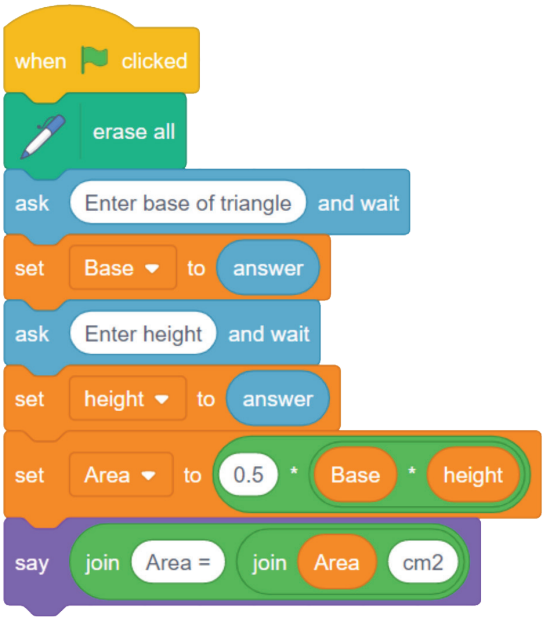


Figure 3.18: Area of a triangle

Calculating volume of shape

To provide more practice, let us do another example following a sequence construct. This time we will write a program to calculate the area of a rectangular prism using Scratch programming.

rectangular prism formula = length × breath × height

The user must enter the length, height and breath of the rectangular prism. The script for the code on the previous page will look like the one illustrated in the figure below:

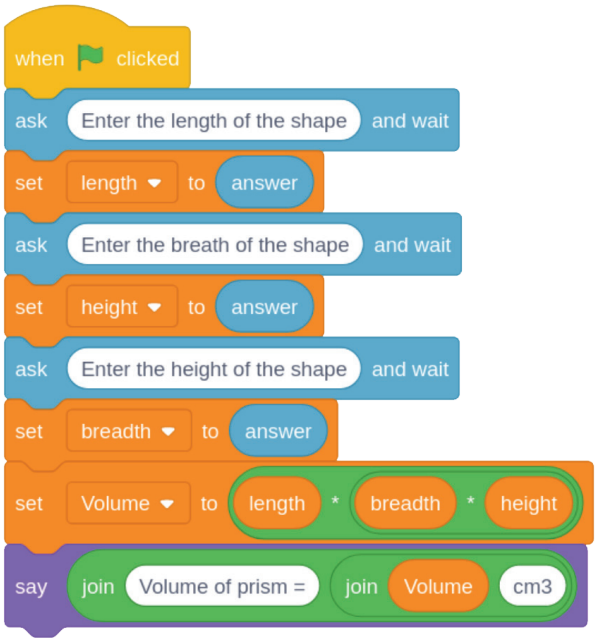


Figure 3.19: Volume of a rectangular prism

You can test and modify the program to calculate any volume if the other two sides are given.

Practical activity 3.1 **INDIVIDUAL ACTIVITY**

Calculating total price including VAT. (15)

Code a program to calculate the total cost of items when the unit price and quantities bought are entered by the user. The total price must include VAT.

NOTE

VAT in South Africa is 15%.

Conditional constructs (*if* and *if-then-else*)

Depending on the condition, Scratch allows you to add as many conditional blocks as you like. In some cases, the number of conditional blocks indicates the complexity of the comparisons being implemented.

Iteration constructs as part of a solution

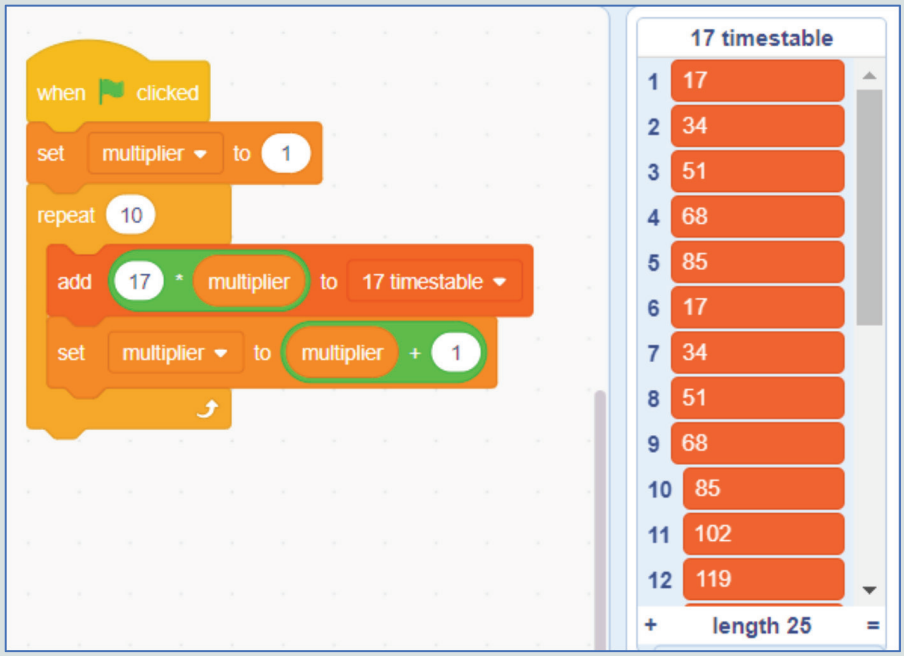
We have already implemented the various iteration blocks such as forever and repeat. These indicate the number of iterations program statements must be executed before a condition or termination. (Refer to the examples given earlier.)

Incorporating a single list

We have looked at lists. The current example uses a list to program a multiplication table.

EXAMPLE 3.2

1. Create a list from the Variables tab and call it *17 times table*.
2. Add a variable and name it *multiplier*.
3. Add a *set multiplier to( )* block and set the value to 1.
4. Insert the *add( )* to block with two operands, as indicated below.
5. Add the *set multiplier to* block.
6. Add a *repeat* block and set it to 10. Slot the *add( ) to* and *set( ) to* blocks into the repeat block.
7. Hit the flag to run the code.
8. Your code must look like the one below.



Times table

Practical activity 3.2 — INDIVIDUAL ACTIVITY

In the figure on the next page, you will notice that the 17-times table was printed, but Scratch did not print it to our liking. We want the times table to be printed as follows:

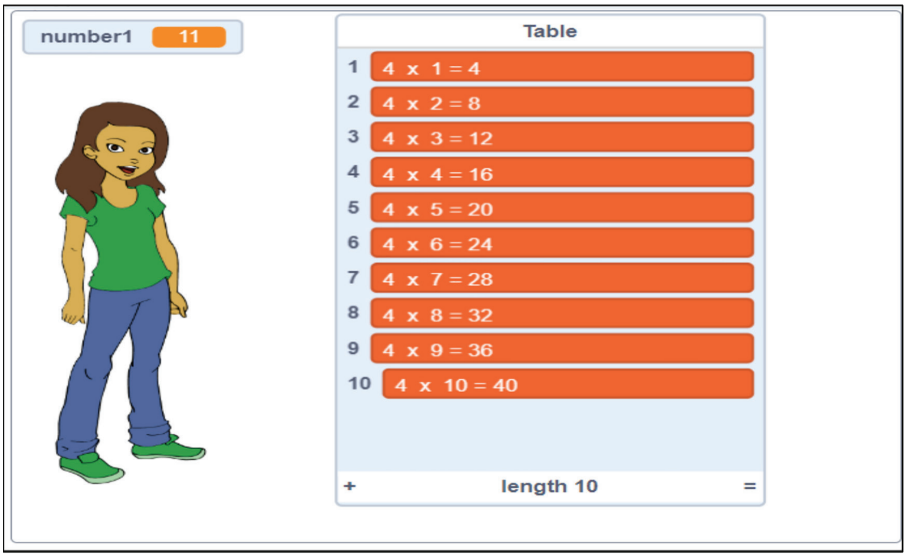
17 × 1 = 17
17 × 2 = 34, etc.

Reprogram your code to give the output for the 4-times table. The output should then read as follows:

4 × 1 = 4
4 × 2 = 8
4 × 3 = 12, etc. up to 4 × 10 = 40

Practical activity 3.2 (continued)

The output should look as follows:



Example of code for times table

String operations as part of the solution

As mentioned earlier, the string function is used to format data into a string. The challenge on times tables included string operations combined with the join block. Strings can also be compared. Comparing strings is the process of analysing and outputting the differences or similarities between two strings.

Comparing strings has many uses, for example:

- checking differences or similarities between a user's response and the correct answer;
- analysing the different responses from users inputted by cloud data on a form project; and
- parsing lines of code.

Scratch uses the ASCI code to compare characters. Upper-case letters (A to Z) are first transformed into lower-case letters (a to z). There is no easy way to distinguish between upper- and lower-case letters.

3.1.7 Programming language tools and constructs

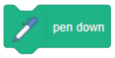
In the preceding lessons, you learned how to move the sprite by using the mouse and keyboard. Now, you will learn how to make your sprite leave a mark while moving. To do this, we will need to activate the pen module, which is not included in the Blocks pallet by default. Navigate to the bottom left corner icon and click  . Select the pen module.

Many blocks are available in the pen module, including erase all, stamp, pen down, pen up, set pen colour, etc. We are going to use a few of these blocks to draw a square as a Ladybug sprite moves when clicked. Start by deleting the default cat sprite and selecting the Ladybug sprite.

Then follow the steps below.

Step 1: With the Ladybug sprite selected, click on Events and add the *when clicked* event handler to start the program.

Step 2: Add a *repeat block* and type in value 4. We are doing this to allow the Ladybug sprite to move four times, as the square has four sides.

Step 3: From the *pen* block, select the *pen down* block 

Step 4: From the *Motion* block, select *move( ) steps*. Edit the value and type in 100 steps.

Step 5: Add a *wait( ) seconds* and edit the value to 3 seconds.

Step 6: Since it is a square, a turn at 90 degrees is needed. Therefore, add the *turn( ) degrees* 

and edit the value to 90.

Step 7: Add a separate script in the same code area.

Step 8: Add the *when clicked* block.

Step 9: From the *Motion* block, add the *go to x: ( ) y: ( )*. Set all values to 0.

Step 10: Add the *point in direction* block and edit the value to 90.

Step 11: Add *erase all( )*.

Step 12: Hit the flag for the first script. The Ladybug will draw a square, as shown in the figure below.

NOTE

The shape is now complete. However, you want to be able to clear the shape at some point in case you want to draw something else. Proceed as follows:

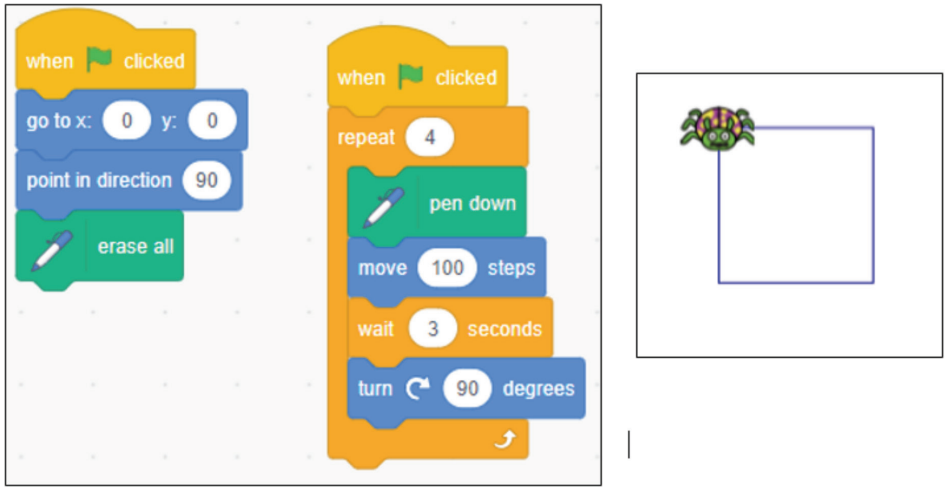


Figure 3.20: Turtle commands using Scratch

If you want to draw a triangle, the code in the figure above needs to be changed as follows:

- The number of iterations from 4 to 3
- The degrees from 90 to 120.

In some cases, if you do not want the Ladybug sprite to be on top of the completed drawing, you will need to add a *pen up* after the repeat block and set *go to x( ) y: ( ) to values away from the shape*.

You can take this a step further and create some nifty drawings. See how the code used in Figure 3.20 has been modified to produce the flower shown on the next page.

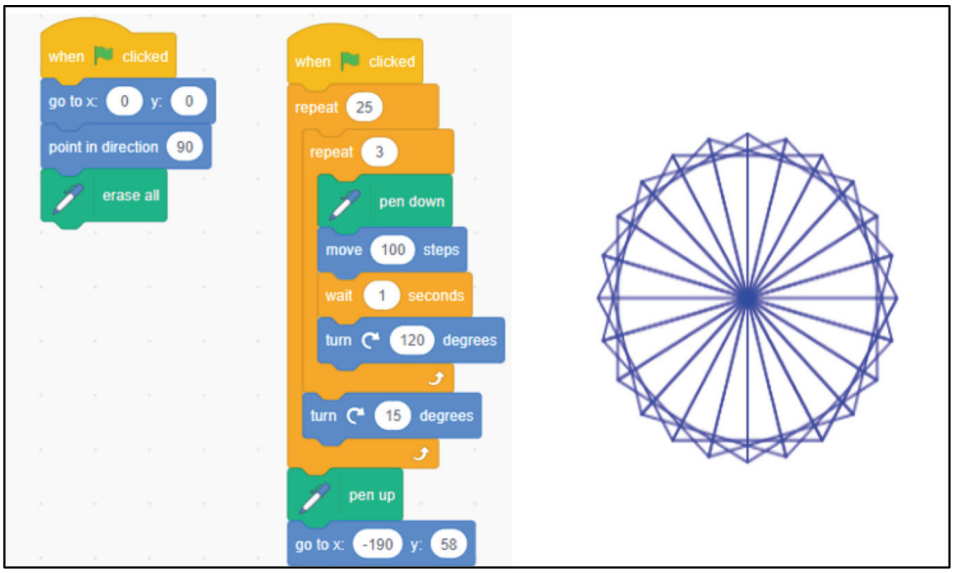


Figure 3.21: Loops of triangles used to produce a flower

Many of the Turtle commands are covered in detail in section 3.4 using Mu IDE with Python.

Sequence

The sequences of commands form one of the first principles of computer programming. Sequence refers to the order in which instructions occur and are processed. They are executed one after the other in the order in which they occur without any condition or repetition.

Selection

Selection is the second basic concept in computer programming. A *selection statement* (also called *conditional statements*) can be described as code that is executed only when certain conditions are met. Selection determines which path a program takes when it is running. It concerns evaluating a condition before executing specific statements. A selection structure is a programming feature that performs different processes based on whether a Boolean condition is true or false. Do you remember the Beetle Butterfly project? The condition was: If Butterfly touches Beetle, the colour must change.

Iteration

Iteration is the third basic concept in computer programming. It is the repeated execution of a section of code (known as a *loop*) when the program is running. The control blocks in Scratch allow for interaction, specifically in the *forever* and *repeat* blocks.

VOCABULARY

Selection – *programming construct where a section of code is run only if a condition is met*

Iteration – *process where the design of a product or application is improved by repeated reviewing and testing*

3.1.8 Devising an algorithm to solve a problem

We are going to use Scratch to calculate simple interest. *Simple interest* is a method used to calculate the amount of interest charged on a sum at a given rate and for a given period.

Simple interest is calculated by using the following formula:

$S.I = P \times R \times T$

Where: P = principal
R = rate of interest in % per annum
T = time, usually calculated as the number of years.

The rate of interest is expressed in percentage r% and is to be written as $r/100$.

- Principal: the amount initially borrowed from the bank or invested; denoted by P
- Rate: the rate of interest at which the principal amount is loaned for a certain time; can be 5%, 10%, 13%, etc.; denoted by R
- Time: the duration for which the principal amount is loaned; denoted by T
- Amount: total owed; the sum of what was borrowed plus the interest.

Simple interest example

Yaseen's father borrowed R1 000 from the bank and the rate of interest was 5%. What would the simple interest be if the amount was borrowed for one year?

Algorithm:

- Start
- Declare variable principal, time and rate
- Accept principle, rate and time
- Calculate simple interest = (principle × rate × time)
- Display simple interest.

Start the Scratch application and apply the code as indicated below.

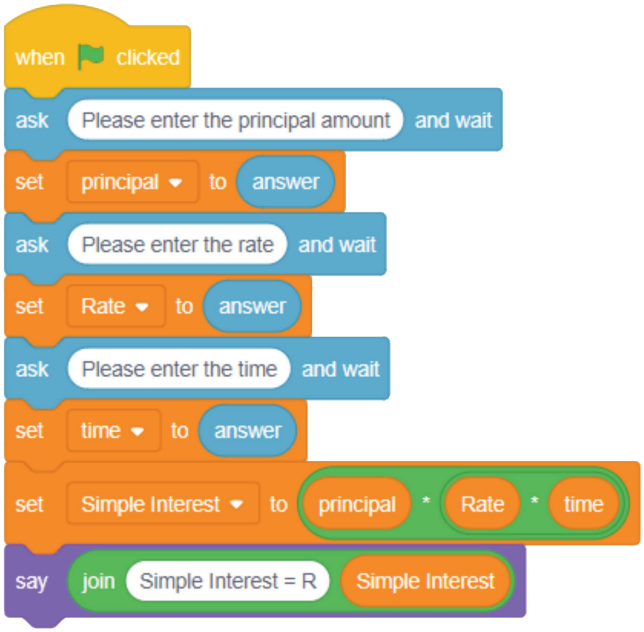


Figure 3.22: Simple interest

Activity 3.1 **INDIVIDUAL ACTIVITY**

1. Define the term *visual programming*. (2)
2. True or false: Scratch is an example of a visual programming language. (1)
3. List THREE advantages of visual programming languages. (3)
4. What is meant by the term *variable* as applied in Scratch programming? (2)
5. Outline the steps needed to create a user-defined variable called *number_1*. (4)
6. What is a *sprite*? (2)
7. List any FIVE blocks found in Scratch. (5)
8. What data type is best suited for a true or false result? (1)
9. GPIO is an acronym used in Raspberry Pi to stand for .... (2)
10. Under which category do you find the *forever* block? (1)
11. Explain the difference between the following programming constructs:
    - **11.1** Sequence
    - **11.2** Selection
    - **11.3** Iteration (3 × 2) (6)

TOTAL: 29

3.2 Install Python on a single-board microprocessor

Python is popular general-purpose, high-level programming language with dynamic semantics. It was created by Guido Van Rossum and released in 1991. It is used in web development, data science, creating software prototypes and many other areas.

3.2.1 Installing Python 3 on the Raspberry Pi

Python is usually installed on Raspberry Pi 4. To check whether Python is installed on your Raspberry Pi, you will need to execute the following command from the Raspberry Pi terminal:

```
pi@raspberrypi:~ $ python --version
Python 2.7.16
pi@raspberrypi:~ $
pi@raspberrypi:~ $ python3 --version
Python 3.7.3
```

Figure 3.23: Checking for Python Version 3

It may be a little confusing, but there are two versions already installed on your Raspberry Pi. When you use the command “python” to run a script, you are running it with Python 2, but the “python3” command will do the same with version 3.

If Python 3 or IDLE is not installed on your computer, follow the installation instructions below for your operating system:

Open a terminal window and type:

```
sudo apt update
sudo apt upgrade
sudo apt install python3 idle3
```

This will install Python 3 (and IDLE). You should then be able to find it in your Application menu. In some cases, you might have to add it to your Raspberry Pi Start Menu.

3.2.2 Installing the Mu Python editor on the Raspberry

Mu is a very simple-to-use Python editor and IDE (integrated development environment) for beginners. It is designed to be as user-friendly and as helpful as possible for new Python programmers. You will install Mu and learn how to use it to create code.

eLINK

~~*[MISSING TEXT]*~~
~~https://docs.python.org/3/library/turtle.html~~

How to install Mu

Mu will work on most operating systems. Just follow the installation instructions for your operating system. Start by updating your Raspberry Pi software:

First navigate to the Raspberry Pi terminal and then type the following command in the terminal.

```
sudo apt-get update
sudo apt-get dist-upgrade
```

Open the *Recommended Software* application from the *Preferences* menu.

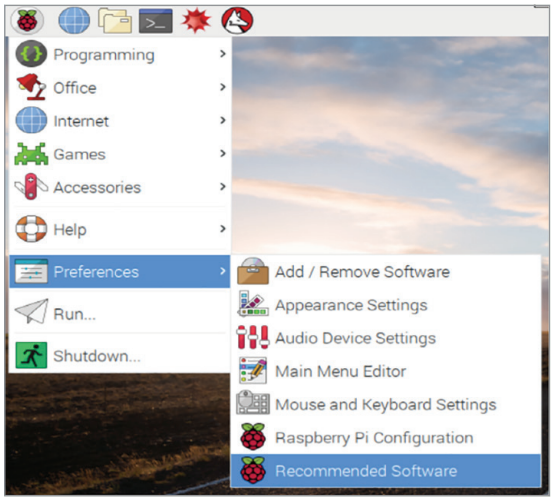


Figure 3.24: Raspberry Pi Preferences

Select *Mu* from the list of applications to install.

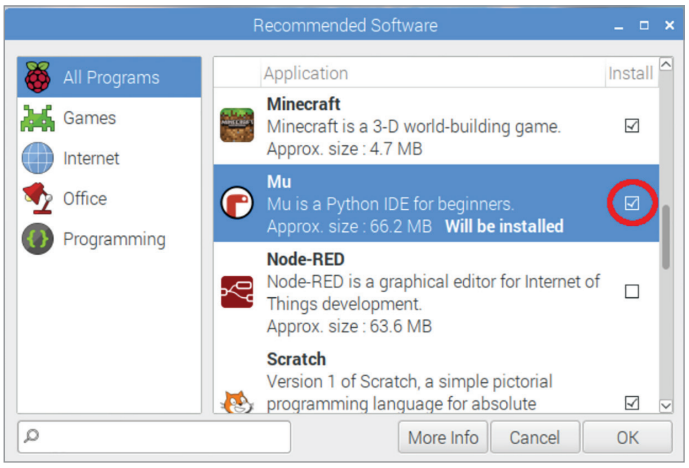


Figure 3.25: Setting up Mu IDE

Click OK to start the installation process.

3.2.3 Start the Mu file editor and create a new Python file

Follow these steps to start Mu:

- Click on the Raspberry Pi Icon ➔ Programming ➔ mu.

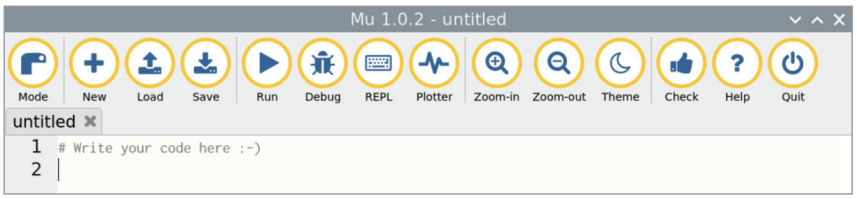


Figure 3.26: Mu screen

The interface looks like the one above.

- Click on the mode icon.
- Select Python 3 and click OK.

3.2.4 Adding *"Hello World"* using Mu IDE

Let us create a simple program to print `"Hello World"`.
In the code space, type the following text:

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

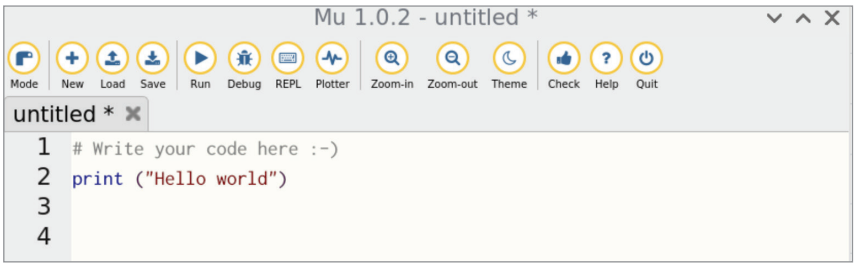
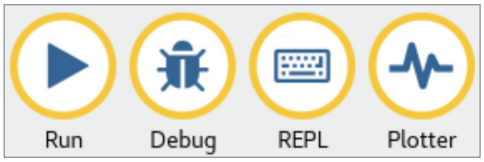


Figure 3.27: Adding text to an open Mu file

The functionality provided by Python 3 mode is contained in the following four buttons:



The Run button does exactly what you would expect. It runs the current script or executes the source code. When this happens, the textual *input* and *output* of the program are displayed in a panel between the text editor and the Mu footer.

Running the above code will show the following output:

```
Hello World
>>>
```

If you type the following statement instead:

```
print('Hello world)
```

You will notice that Python underlines the text in red to indicate an error. The resultant output will be:

```
print('Hello world)
^
SyntaxError: EOL while scanning string literal
>>>
```

Python indicates an error because it does not know where to stop printing. To fix the code, as in the previous example, you must add a corresponding quote symbol. However, if you type: `print(3)` and hit `Run`, no error will be generated. That is because the interpreter recognises it as a number.

When the code is running, the `Run` button turns into a `Stop` button. Click *Stop* to force your code to exit cleanly. If your code finishes running before you click Stop, you will be presented with a Python prompt that allows you to type Python commands to check the end state of your program. When in doubt, just click *Stop*.

NOTE

The output section displays: `Hello world`.

As these are comments, the text `"#Write your code here:-)"` *has been omitted from printing. Comments will be discussed later.*

The `REPL` button opens a new panel between the text editor and Mu footer. In layman's terms, REPL stands for "Read, Evaluate, Print, Loop", which is a concise description of what the panel does for you. It reads instructions that you type in Python, evaluates the meaning, prints any results it has for you and then loops back to wait for your next Python command.

When you click on run, the Mu editor asks you to save the file. You can navigate to any folder of your choice on your Raspberry Pi or the external storage and click Save. The file will be saved with a name you assign and a .py extension, indicating that it is a python script file.

Task 3.1

Save the file with a name *helloworld.py* in a folder called *Python_Projects*, which you create in documents.

Open an existing file

You must know where the file you have saved earlier is stored to open it. Click the Load icon and navigate to the location of your saved file. In the previous example, we saved *helloworld.py* in the Python_Projects folder, so now we are going to open it.

Add the following two lines of code to the *helloworld.py* file:

```
#Write your code here
first_name = "Leandre"
print("Hello" + first_name)
```

In the previous example, you will have noticed that the output is the string that was enclosed in double quote symbols. As explained previously, a *string* is a combination of letters and/ or numbers in quotes. Single quotes (' ') can be used, but make sure you are consistent.

Our program runs and displays the following output below the code area:

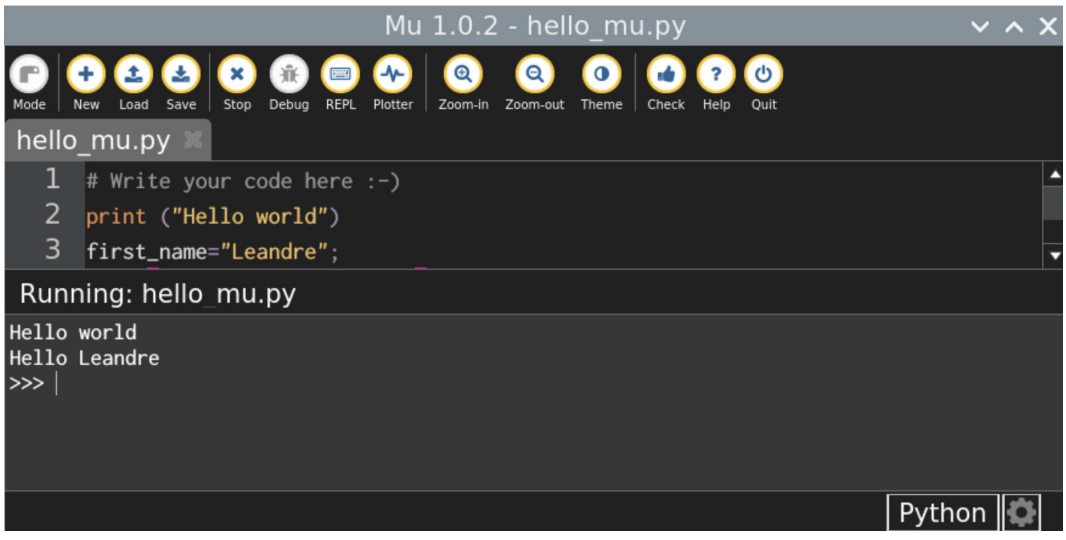


Figure 3.28: Variables

In the above example, the word *first_name* is a variable. A variable, as discussed earlier, is a name you give to a computer memory location that is used to store a value in a computer program. A variable will have a name and a value.

Open and run/execute simple sample files

Practical activity 3.3 **INDIVIDUAL ACTIVITY**

Using Mu

Create a Python program using Mu. The program must be able to do the following:

1. Print the following statement: "My firstname is:"
2. Assign a name into a variable called first_name.
3. Display the value stored in the variable.
4. Print the following statement: ("My firstname is:")
5. Assign a name into a variable called last_name.
6. Print the value stored in the variable.
7. Print the *fullname* in a sentence as follows: first name and last name.
8. In the case of this example, the string *Leandre* was assigned to *first_name* and *Ngele* to last_name. Please use your own details.
9. Display the text as "My fullname is:"
10. Save your file with a name called *my_details.py*.

Running the program should produce the following output:

```
Running: my_details.py
My firstname is :
Leandre
My surname is :
My fullname is : Leandre Ngele
>>>
```

Output using variables

Practical activity 3.4 **INDIVIDUAL ACTIVITY**

We will now try to work with numbers. You need to create a new file called *numbers.py*. Paste the following code into the editor and run it.

```
#Write your code here :-)
number1 = 67
number2 = 55
print(number1 + number2)
```

What is the output?

Activity 3.2 **INDIVIDUAL ACTIVITY**

1. What command would you use on a Raspberry Pi to check whether Python is installed? (2)
2. What does *print statement* do in Python? (2)
3. True or false: Python is case-sensitive. (1)

TOTAL: 5

3.3 Basic Python applications and the Turtle library

Turtle is a Python library that is used to create graphics, pictures and games. It was developed by Wally Feurzeig, Seymour Papert and Cynthia Solomon in 1967 as part of the original Logo programming language.

3.3.1 Differentiating between a *compiler* and an *interpreter*

Both compilers and interpreters are used to convert a program written in a high-level language into machine code understood by computers. However, there are differences between how an interpreter and a compiler work.

A *compiler* will convert the code into machine code (create an executable) before the program is run. *Interpreters* convert the code into machine code when the program is run. Interpreters convert code line by line, whereas compilers translate the whole program at once. Python is an interpreted language. Examples of compiled languages are C#, Java, VB.NET, etc.

3.3.2 Differentiate between a shell and an IDE

The Python shell is a command-line tool that starts up the Python interpreter. You can test simple programs and write some short programs. The shell is a basic read-eval-print loop (REPL). It reads a Python statement, evaluates the result of that statement and then prints the result on the screen. Thereafter, it loops back to read the next statement. When you open a MU IDE, there is an REPL button that will start the shell. You can also start the shell from the terminal by typing *python*.

However, to write a more complex Python program, you will need an editor. An IDE is a program dedicated to software development. As the name implies, IDEs integrate several tools specifically designed for software development. These tools usually include an editor designed to handle code (with, for example, syntax highlighting and auto-completion functions).

3.3.3 Characteristics of Python

The following are characteristics of Python:

- Easy to code – it allows users to code in a variety of ways
- Free and open source – Python language is freely available on the official website
- Supports object-oriented language and concepts of classes, objects encapsulation, etc.
- Portable language/machine-independent
- Interpreted language, as Python code is executed line by line
- Includes many libraries for things such as regular expressions, unit testing, and web browsers.

3.3.4 Set up a simple Turtle Python program

Turtle is a Python feature that allows us to create pictures and shapes by providing us with a virtual canvas and an onscreen pen. We can move the turtle in four directions. Turtle is an integrated part of the Python package and does not need to be installed separately.

The four basic functions are as follows:

- Forward – moves the turtle to the right
- Backward – moves the turtle backward (left)
- Left – causes a rotation in an anticlockwise direction
- Right – causes a rotation in a clockwise direction.

Using functions such as turtle.forward( ) and turtle.right( ), we can move the turtle around. Let us create a program called *my_drawing* and add the following lines of code.

```
#drawing
import turtle
myscreen=turtle.Turtle( )
```

Upon running the program, you will see a screen similar to the one in the figure below.

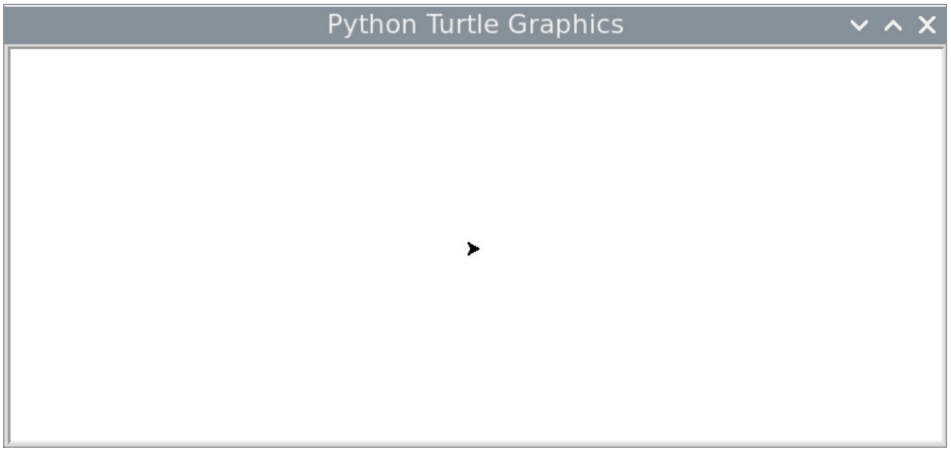


Figure 3.29: Turtle output screen

The turtle (which is the small arrow on the screen) can move forward and backward in the direction that it is facing.

3.3.5 The purpose of the *import* statement in Python

To use *Turtle* methods and functionalities, we need to import Turtle. We can execute Turtle programs in four steps:

- Import the turtle module
- Create a turtle to control
- Draw using the turtle methods
- Run turtle.done( ).

3.3.6 Drawing a line using the forward function

The following is an example of a program used to draw a straight line. Type the following in the Mu IDE.

```
#Draw a straight line using turtle
import turtle
myline = turtle.Turtle( )
myline.fillcolor("cyan")
myline.forward(150)
myline.hideturtle( )
turtle.done( )
```

Run the program. You will notice that a straight line is printed.

Let us get to understand the code.

- Line 1 is just a comment, so it will not be printed.
- Line 2 imports the library for the drawing.
- Line 3 allows you to create an object so that it can be used on your screen.
- Line 4 adds colour to your line. Change the colour and see what happens.
- In line 5: The turtle is instructed how many steps to take in the right direction. Use backward and see what happens.
- Line 6 will hide the turtle after it is drawn.
- Line 7 indicates that nothing is following.

The output will be as follows:

Figure 3.30: Straight line

Task 3.2

What are the TWO ways in which to make the same line double its initial size?

The most common turtle methods are listed in the table below.

METHOD	WHAT TO INCLUDE IN BRACKETS	DESCRIPTION
Turtle( )	None	Creates and returns a new turtle object
forward( )	Amount	Moves the turtle forward by the specified amount
backward( )	Amount	Moves the turtle backward by the specified amount
right( )	Angle	Turns the turtle clockwise
left( )	Angle	Turns the turtle anticlockwise
penup( )	None	Picks up the turtle's pen
pendown( )	None	Puts down the turtle's pen
up( )	None	Picks up the turtle's pen
down( )	None	Puts down the turtle's pen
color( )	Colour name	Changes the colour of the turtle's pen
fillcolor( )	Colour name	Changes the colour of the turtle will use to fill a polygon
heading( )	None	Returns the current heading
position( )	None	Returns the current position
goto( )	*x*, *y*	Move the turtle to position *x*, *y*
shape( )	Shape name	Should be 'arrow', 'classic', 'turtle' or 'circle'

Table 3.2: Common methods used to create drawings

3.3.7 Turn using the right and left functions

- Right(angle) or turtle.right(angle in degrees) – this method moves the turtle right by angle units.
- Left(angle) or turtle.left(angle in degrees) – this method moves the turtle left by angle units.

For example, let us modify our earlier program to draw a line.

```
#Turning
import turtle
myline = turtle.Turtle( )
myline.fillcolor("cyan")
myline.forward(150)
myline.left(90)
myline.forward(150)
myline.hideturtle( )
turtle.done( )
```

In the above example, the code demonstrates how to draw a straight line and turn anticlockwise at 90 degrees.

3.3.8 Using left and right functions

If you want to draw a square, extend the line by turning and moving the same distance four times. Note that you only need to turn three times. The screenshot below shows the instructions for drawing a square.

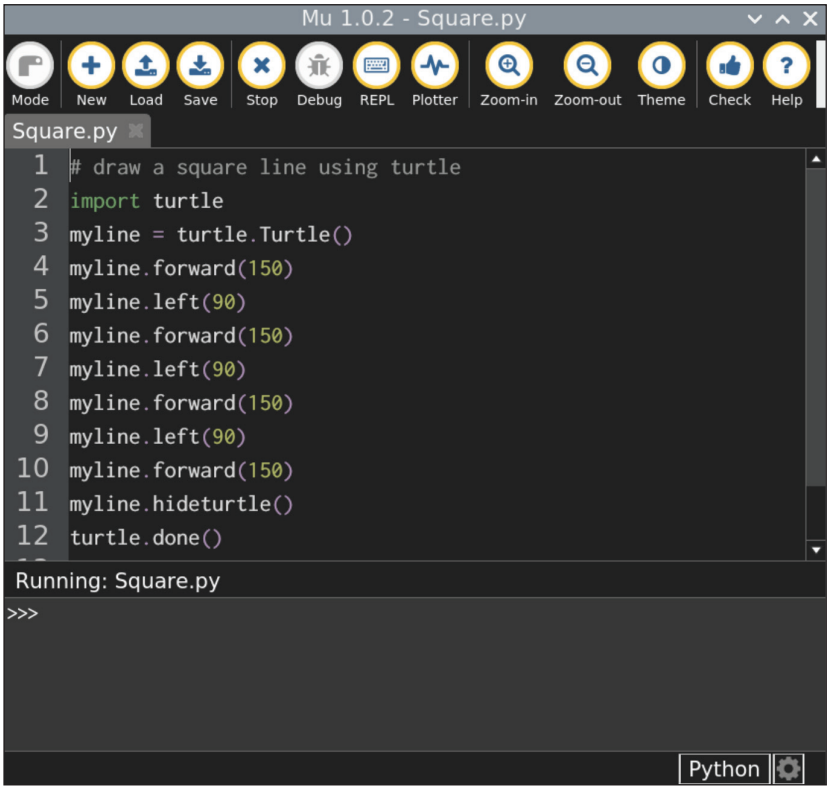


Figure 3.31: Drawing a square

A square will appear on the screen if you run the code on the previous page.

Let us try to draw the letter 'T'. Here we will use the penup( ), pendown( ), forward, backward, left, right and pensize( ) methods.

```
#Writing a letter t
import turtle
turt=turtle.Turtle( )
turt.pensize(10)
turt.penup( )
turt.fd(120)
turt.right(90)
turt.pendown( )
turt.fd(100)
turt.bk(100)
turt.left(90)
turt.fd(40)
turt.bk(80)
```

eLINK

The link below explains how to draw letters using the Turtle library:
https://youtu.be/C1HNStRVKPw

Activity 3.3 **INDIVIDUAL ACTIVITY**

Create a Python program using Mu editor to draw the following:

1. A triangle (9)
2. A circle (6)
3. The letter H. (15)

TOTAL: 30

3.3.9 Colours in Scratch

Python recognises a large number of colour names. It also accepts a hex code (code that has six characters) that describes how to mix different amounts of red, green and blue to produce a specific colour. The code must be followed by a # character.

As an additive colour model, RGB combines the red, green and blue primary colours of light in various ways to reproduce a wide range of colours. The name of the model is derived from the initials of the three additive primary colours, red, green and blue.

3.3.10 Changing the line colour using the colour function

The Python Turtle module provides us with many functions that allow us to add colour to the shapes or objects we draw. To change the colour of the lines, use the following:

```
turtle_name.color('color_name')
```

`begin_fill( )` – This method is used just before drawing a shape to be filled. Once applied the shape to be drawn will be filled with the chosen colour, i.e. use the syntax:

```
turtle.begin_fill( )
```

To terminate the filling process, use the following syntax:

```
turtle.end_fill( )
```

The code below demonstrates how to draw a square and fill it with red colour. Retype the code on your Raspberry Pi and run the program to see the real output.

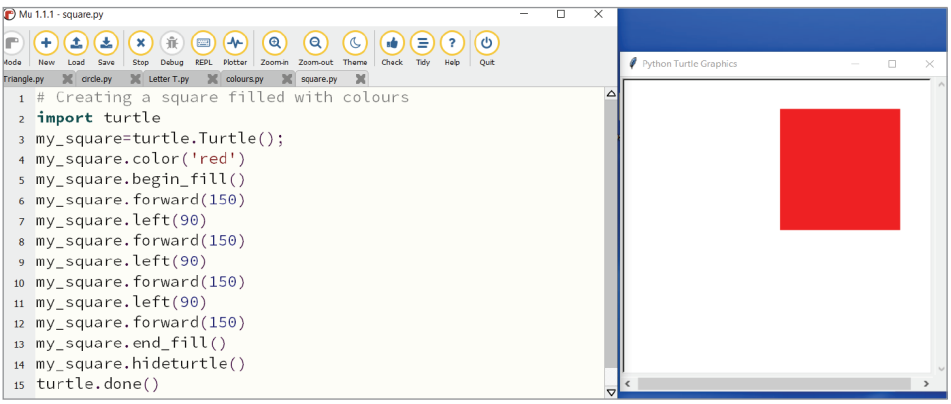


Figure 3.32: Implementing colours

3.3.11 Combine drawing with changing the colour

In this section you will learn more about iteration. We are going to use a loop for sequential traversal. We want to draw a star like the one shown below:

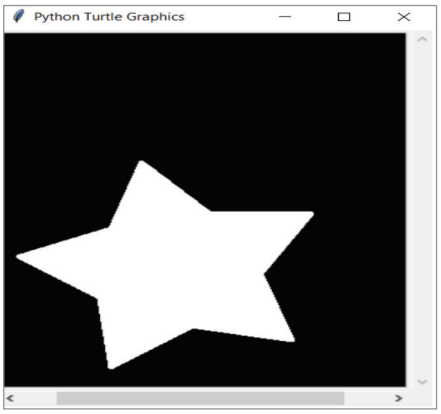


Figure 3.33: Drawing a star

The background has been set to black. The syntax to use is:

```
turtle.bgcolor('black')
```

To set the fill colour of the star, we use the following syntax:

```
turtle.color("white")
```

The steps are set out here:

```
#importing package
import turtle
#size of star
star_size = 80
#object colour
turtle.color("white")
turtle.bgcolor('black')
#object width
turtle.width(4)
#angle to form star
angle = 120
#colour to fill
turtle.fillcolor("white")
turtle.begin_fill( )
```

Summative assessment

1. Define the term *block-based programming*. (2)
2. List any TWO examples of block-based programming languages. (2)
3. List THREE disadvantages of block-based programming languages. (3)
4. What is an *integer data type*? Give an example. (2)
5. Use Scratch and Raspberry Pi to create a program to simulate traffic lights.
   The lights should turn on/off in the sequence: green, orange, red. (10)
6. What is the difference between a *global variable* and a *local variable*? (4)
7. Differentiate between the terms *interpreter* and *compiler* as used in programming. (4)
8. What will the following expression evaluate to? (2)

9. The following table shows methods used in Turtle to create images on screen. Pair the method with the correct description. Select the appropriate method from the ones listed below. Write down only the question number and the correct method, e.g. 9.1 color( ).

forward( ), pendown( ), right( ), Turtle( ), left( ), up( ), backward( ), penup( ), down( ), fillcolor( )

DESCRIPTION	METHOD
1. Creates and returns a new turtle object	
2. Moves the turtle forward by the specified amount	
3. Moves the turtle backward by the specified amount	
4. Turns the turtle clockwise	
5. Turns the turtle anticlockwise	
6. Picks up the turtle's pen	
7. Puts down the turtle's pen	

(7 × 1) (7)

TOTAL: 36

Module 4

Programming tools and utilities

After you have completed this module, you should be able to:

- define the term *revision control* in general;
- explain the consent of a revision number;
- name and explain the two main components of a software revision (timestamp and author);
- explain why there is a need for a logical way to organise and control revisions in software development;
- define the term *version-control system* (VCS);
- elaborate how a VCS differs from *generic revision control*;
- differentiate between a *branch* and a *trunc* as used in VCSs;
- list advantages to splitting development of software into different branches;
- list the disadvantages to keeping multiple application-version copies in separate folders when developing software;
- explain the concept of a *centralised VCS*; and
- explain the concept of a *distributed VCS*.

Introduction

Within developer environments, software management is a critical procedure. The process of tracking and managing changes to source code is known as *source-code management*. Source-code management provides a complete history of all modifications made to a shared codebase for developers and other stakeholders. This guarantees that developers are working with the most recent code and that no conflicts exist. There is often confusion between the terms *source control*, *revision control* and *version control*. This module clarifies these terms. The figure below provides a quick overview of source-code management.

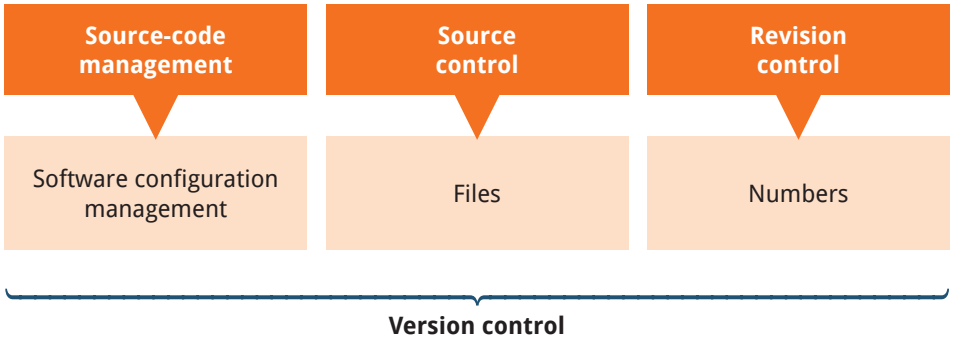


Figure 4.1: Version control

4.1 Source control

4.1.1 Defining *revision control* in general

VOCABULARY

Revision control – *class of systems responsible for managing changes to computer programs, documents, large websites or other collections of information*

Rollback – *process of returning a software program to an earlier version*

As you know, software is regularly updated, and the versions are made generally available to users. However, some form of control is necessary. **Revision control**, also known as *version control* and *source control*, is a system that records changes to a file or set of files over time so that you can recall specific versions later. In other words, it is a system that records any changes made by developers.

Revision-control methods (further discussed in section 4.1.4) are built into several applications, such as spreadsheets and word processors. Designers and developers use revision control to keep track of their documents and the configuration files of their projects. With the correct use of revision-control systems (RCSs), high-quality documentation and products are possible.

The following are characteristics of an RCS:

- Check-ins and check-outs are possible for each document that is kept.
- The system can be used to identify and fix side features and defects in a streamlined manner.
- Its tag system aids in the distinction between alpha, beta and release versions of various documents or apps.
- RCS must allow for rollback or fix-forward. **Rollback** describes the process of returning a software program to an earlier version after encountering issues with the later version. The opposite is *fix-forward*. These are systems that allow for future developments and easy integration.

4.1.2 The concept of a revision number

Each software version has a name. The names of these versions are usually assigned a number or letter code. There are several standard methods of naming software.

Some of the most common ways to number software versions are the following:

- Semantic numbering – a three-digit numbering technique based on *Major.Minor.Patch* (the first number indicates major version changes; the second, minor version changes related to backward-compatible additions or new functionality; and *patch* refers to a set of changes designed to update, fix or improve the program)
- Date of release – software version numbers are based on the release date, e.g. 22.02 (Feb 2022)
- Unary numbering – with only one operand
- Alphanumeric codes – may consist of alphabetic characters and numbers, e.g. r2.1
- Sequential numbering – follow a sequence, e.g. 2.1, 2.2, 2.3, etc.

Each software revision is associated with a timestamp and the person making the change. Revisions can be compared, restored and, in some types of files, merged.

4.1.3 Two main components of a software revision

VOCABULARY

Timestamp – *sequence of characters or encoded information identifying when a certain event occurred*

Software author – *person who has authored or participated in the development of the software or any portion thereof*

Two common components of software revisions are **timestamp** and **software author**.

- *Timestamps* refer to a sequence of characters or encoded information that tells us when an event occurred, usually indicating the time and date. It is not necessary to use an absolute notion of time when simulating timestamps. The following is an example of a timestamp: 2022-05-30 T 10:45 UTC.
- *Software author* revision is an author-based selection feature that enables users to specify a truly arbitrary set of revision items. For each revision of a baseline item, authors can use their names or specify a symbolic name of their choice. When a single name or symbolic name is assigned to all items that make up a certain compatible system, all those items can be checked by using that name.

Software revision control best practices

Software development includes the continuous process of modifying programs. Revision- or version-control systems (VCSs) make this task easier.

The basic rules that must be considered when changes are made include the following:

- Remain transparent and consistent
- Publish a release schedule to avoid surprises
- Explain the versioning system to the users
- Inform users what is new
- Ask users for their opinion.

4.1.4 The need for organised and controlled revisions

Version control is used to track and control changes to source code. It is necessary for the following reasons:

- By sticking to one method of numbering, users can stay organised and keep track of updates and releases of a particular piece of software.
- Organising revisions allows developers to work on the same parts of a project at the same time without worrying about overwriting each other's contributions.
- From an organisational perspective, managing revisions gives a company an edge to compete with other software development houses.
- By logically organising revisions, developers can also easily identify differences between different versions of files when troubleshooting an issue.

4.1.5 Defining *version-control system*

VOCABULARY

Version-control system *– system that tracks changes to a file or set of files over time*

As explained earlier, version control is used to track and control changes to source code. **Version-control systems** (VCSs) automatically maintain character-level changes for all files stored within the system, allowing a complete retrace of all versions of each file, the author of those versions and a complete rollback of the changes from the beginning of version control.

An RCS is an essential tool for any organisation with multiple projects, as it can identify issues and bugs. Furthermore, it can store, log, identify and merge information related to the revision of software. It allows the retrieval of an earlier working version of an application or document whenever required.

Developers are increasingly using VCS to keep track of source code and project files that are under development and thereafter to retrieve past versions. VCS allow multiple developers, designers and team members to work together on the same project at the same time. It is common for software developers to make continual changes to files and codes that add or remove features during the development process. Because the number of revisions grows, it is difficult to manage and organise the codes and files. Therefore, the existence of a VCS can help software developers to simplify and speed up the development process.

Common features of a VCS

The following are common features of VCS:

- Backup and restore – files can be backed up as they are edited and have a rollback facility
- Synchronisation – allows sharing source code and updating with latest revisions
- Track changes – reasons for changes can be recorded
- Track ownership – person who has made changes (commits) can be tracked
- Sandboxing – involves isolated changes during testing or before finalisation
- Merging – files can be merged after changes.

4.1.6 How a VCS differs from generic revision control

The idea of VCS was developed in the 1980s by Walter F. Tichy. This automated the process and made storing, retrieval, logging, identification and merging of file revisions easier.

The disadvantages of a generic RCS include the following:

- It operates only on single files – has no facility for handling entire projects or directories
- The command syntax is cumbersome
- It lacks a concurrency feature that allows developers in different places to work on the same project.

VCS introduced new concepts to versioning in several ways. It uses a client/server model, supported branch imports, unreserved checkouts and symbolic mapping. The only challenge, however, is that it requires a network to run. VCS are most run as stand-alone applications, while revision control is embedded in various types of software.

Benefits of VCS

VCS have the following benefits:

- Improving the speed of project development by providing efficient collaboration
- Using better communication and assistance to boost productivity, streamline product delivery and develop developers' skills
- Reducing the possibility of errors and conflict during project development
- Allowing developers from different geographical locations to contribute to the project
- Maintaining a different working copy for every contributor, and the working copy cannot be merged to the main file unless it is validated
- Assisting in the recovery process in the event of a disaster or contingent situation
- Providing information about who, what, when and why changes have been made.

VCS can be categorised into three types: centralised, distributed and local.

- Centralised systems: all files are stored in one repository; examples include Subversion, CVS and Perforce
- Distributed systems: files are stored across multiple **repositories**; examples include Git, Mercurial, Bazaar and Darcs
- Local systems: involve copying files into another directory (such as a time-stamped directory); simple yet error-prone – it is easy to forget which directory you are in and accidentally write to the wrong file or copy over files.

VOCABULARY

Repository – *database of changes*

Because Git is extremely popular, we will discuss it in more detail.

Git history

Git is an open-source, free VCS that can handle everything from very small to large-scale projects with ease. Since its was launched in 2005, Git has become one of the best version-control tools available.

How Git works

Git considers its data to be like a series of snapshots of a tiny file system. When you commit or save your project using Git, it effectively takes a snapshot of what all your files look like at that time and keeps a reference to that snapshot. To save space, Git stores only a link to the last identical file it has already saved.

Git has four main states in which files can be saved: working directory, staging area, local repository and central repository.

- Working directory: where projects are created (code written) and changes are made
- Staging area: where code can be reviews before making a commitment
- Local repository: where you may commit changes to the project before pushing them to the central GitHub repository
- Central repository: where the primary project is stored (a copy is kept locally by each team member).

Features of Git

- Provides strong support for non-linear development
- Complete database is mirrored on every developer's computer (i.e. it is a distributed repository model)
- Compatible with existing systems and protocols such as HTTP, FTP and ssh
- Can efficiently handle small to large projects
- Pluggable merge strategies
- Toolkit-based design.

Advantages of using Git as a distributed version-control system (DVCS)

- Super-fast and efficient performance
- Cross-platform use
- Code changes can be easily and clearly tracked
- Easily maintainable and robust
- Offers an amazing command-line utility known as *git bash*
- Provides a GIT GUI, where you can rescan, change states, sign off, commit and push the code very quickly.

Disadvantages of Git as DVCS

- Complex and bigger history log can become difficult to understand
- Does not support keyword expansion and timestamp preservation.

4.1.7 Differentiating between a branch and a trunk as used in VCSs

Working with version control requires understanding of key terms such as repository, branching, trunking, and merging. A *repository* stores metadata for a set of files or a directory structure. In a distributed or centralised VCS, the repository may be maintained on a single server or replicated on every user's system. A repository contains all the edits and historical versions (snapshots) of the project.

VOCABULARY

Branch – *instruction that can cause a computer to begin executing in a different instruction sequence*

Branching

A **branch** is an instruction in a computer program. It is a subdevelopment area where parallel development on different functionalities takes place. It can cause a computer to begin executing a different instruction sequence, deviating from its default behaviour of executing instructions in order. *Branching* means you diverge from the main line of development and continue to do work without messing with that main line. Branches are also called *code lines* or *streams*.

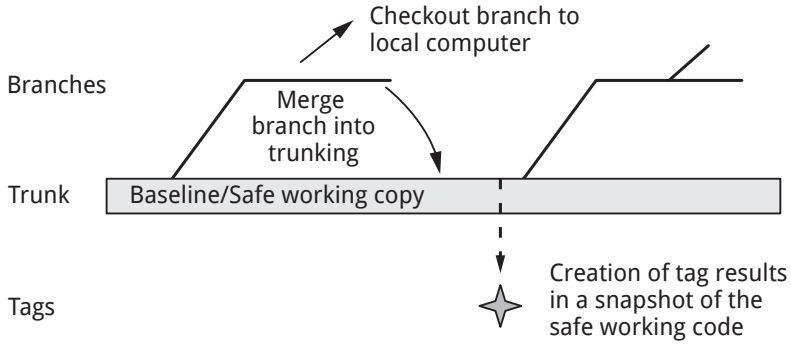


Figure 4.2: Branches and trunks

Trunk

VOCABULARY

Trunk – *unnamed version of a file or program that is being processed under revision control*

A **trunk** is the main development area – the base code into which all subsequent code is merged. A trunk-based development practice integrates small, frequent updates to a core branch. Source code is copied outward from the trunk as branches (refer to the diagram above). Trunking enables a development team to avoid lengthy code-integration processes and boost productivity. The trunk is pushed with all changes, even those not in final form. In the event of an error, it will be addressed and resolved as soon as possible.

Trunking has the following advantages:

- Speeding up reviews
- Avoiding code drift
- Fostering collaboration
- Promoting faster integration.

4.1.8 Advantages of splitting development of software into different branches

There are many advantages to splitting development of software into different branches.

Branching:

- helps developers to simplify procedures, which improves overall workflow efficiency;
- allows developers to collaborate in continuous parallel development;
- ensures that the main version of the content stays in a deployable state (i.e. releases can take place while development is ongoing); and
- provides additional flexibility and control over what is generated or produced for a project.

4.1.9 Disadvantages of keeping multiple application version copies in separate folders when developing software

Using a file-based system to keep organisational information has the following disadvantages:

- Data redundancy – can lead to inconsistency in data format, wastage of storage space and organisational time

- Data isolation – makes it difficult for new applications to retrieve the appropriate data, which might be stored in various files
- Security problems – application requirements are added to the system in an ad hoc manner, so it is difficult to enforce constraints.

4.1.10 Centralised version-control systems

We referred briefly to centralised version-control systems (CVCSs) in section 4.1.6. A *CVCS* is a development approach that allows developers to collaborate with each other. The master copy of the file history is stored there and, in order to read it, retrieve it and make changes to it, users must contact the server. A CVCS consists of a central server containing all the files and many clients that are able to check out files from it. This model has been around for many years, but it has one point of failure that can disrupt the process. If that server is down for an hour, nobody can collaborate at all or save versioned changes to anything they are working on. If the hard disk of the central database is corrupted and no backups have been made, you lose everything (except snapshots of individual machines locally). The figure alongside shows how a CVCS functions.

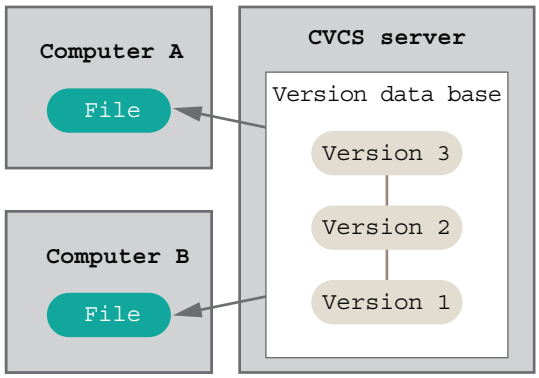


Figure 4.3: CVCS

Advantages of CVCS

The advantages of CVCS include the following:

- Reasonably easy to set up
- Various options (proprietary and open source) available
- Allows for file sharing among team members
- Project is stored on a more reliable server (possibly a cloud)
- Admin can control the use and structure of the repository.

Disadvantages of CVCS

The disadvantages of CVCS include the following:

- Single point of failure (if the server fails, changes will not be available)
- File conflicts due to updates from different people.

4.1.11 Distributed version-control systems

DVCSs were briefly mentioned in section 4.1.6. Today, open-source software projects largely employ DVCS. This is due to the risk associated with local VCS and CVCS (as listed above).

Multiple repositories are present in DVCS. Every user has his or her own repository and working copy. Simply committing your modifications does not give anyone access to them. This is because committing will only reflect changes in your local repository. You must push the changes to the central repository to make them available in the others.

The following four things are required to make your changes visible to others:

- Commit the changes
- Push changes to the remote repository
- Pull from the remote repository to get latest changes
- Update.

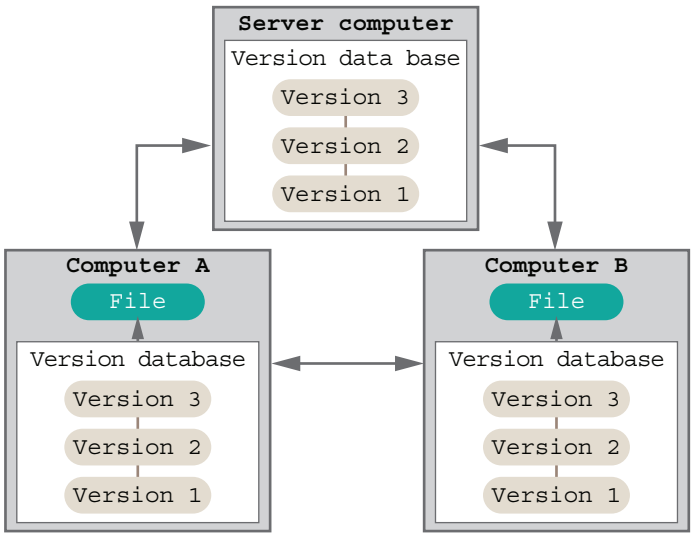


Figure 4.4: DVCS

The diagram above shows how a DVCS functions in both directions by storing the entire history of the files on each machine and also allowing users to sync changes to the server whenever required so that all users can benefit from the changes. A good example of a DVCS is Git and GitHub. GitHub is a code-hosting platform for collaboration and version control. GitHub lets you (and others) work together on projects.

Advantages of DVCSs

- Reliable (everyone has a copy of all versions)
- Allows file share among team members
- Various options available.

Disadvantages of DVCSs

- More complex to use/set up
- Burden on local storage.

A summary of the comparison between a CVCS and a DVCS is shown in the table below.

VCS AREAS	CVCS	DVCS
Repository	Relies on one central repository, which is the server	Every user has a complete repository
Repository access	Requires network connection	Allows every user to work completely offline and connect to network when sharing/pulling resources
Examples of tools	Subversion, Perforce	Git Mercurial, BitKeeper
Characteristics of suitable projects	• Projects that require teamwork • Team must be located in a single site	• Can be applied to small or large projects • Team can be located in multiple sites

Table 4.1: Comparison of VCS

eLINKS

Visit these links to learn more about version control and Git:

- https://git-scm.com/video/what-is-version-control
- https://git-scm.com/video/what-is-git
- https://git-scm.com/video/get-going

Activity 4.1 — INDIVIDUAL ACTIVITY

1. Explain the term *revision control*. (2)
2. List FOUR characteristics of revision control. (4)
3. … describes the process of returning a software program back to an earlier version after encountering issues with the later version. (1)
4. Name the TWO main components of a software revision. (2)
5. Explain why there is a need for a logical way to organise and control revisions in software development. (2)
6. List any FOUR common features of version control. (4)
7. Differentiate between the terms *branch* and *trunk* as used in VCS. (2)
8. True or false: Git is an example of a DVCS. (1)
9. True or false: A CVCS is a development approach that allows developers to collaborate with each other. (1)
10. The folder in which a user creates the project (writes code) and makes changes to it is called a …. (1)
11. Subversion in an example of a/an …. (1)

TOTAL: 21

Summative assessment

1. Name THREE types of VCS. (3)
2. List FOUR common ways of numbering software versions. (4)
3. Explain the concept of *major.minor.patch* as applied in semantic versioning. (6)
4. Explain what is meant by *timestamp* in relation to software revision. (2)
5. List any TWO drawbacks of RCS. (2)
6. True or false: Git is a cross-platform DVCS. (1)
7. Explain each of the following concepts as used in Git:
   - **7.1** Working directory
   - **7.2** Staging area
   - **7.3** Local repository
   - **7.4** Central repository (4 × 2) (8)
8. With the aid of a diagram, explain the concept of a DVCS. (5)

TOTAL: 31

Module 5

Introduction to a high-level programming language

After you have completed this module, you should be able to:

- differentiate between *lexis* and *syntax*;
- differentiate between an *instruction* and a *comment*;
- define the term *interpreter* and elaborate on the function the interpreter serves;
- explain how an *interpreter* differs from a *traditional compiler* as used by C/C++;
- explain Python's traditional runtime execution model (source to bytecode to runtime);
- explain why Python programs use the *.py* extension;
- write the Python code to display output "Hello World" using a basic print statement in the interactive Python prompt;
- create a new project using the high-level integrated development environment (IDE);
- write the Python code to display output "Hello World" using a basic print statement in PyCharm;
- list and explain two types of comments allowed by the Python interpreter;
- explain the term *block-structured language* and how it applies to Python;
- discuss how a block of code is defined in Python using whitespace indentation;
- explain how the whitespace indentation in Python differs from the braces { } used by other programming languages; and
- investigate the IDE message window to locate line numbers that contain error.

Introduction

A *high-level language* is a programming language that allows a programmer to create programs that are independent of the machine being used. High-level languages get their name from the fact that they are more like human languages than machine-level languages. Python is an example of a high-level programming language that may be used to write object-oriented, structured and functional code (as you learned in Module 3). The figure below depicts an abstraction of computer languages.

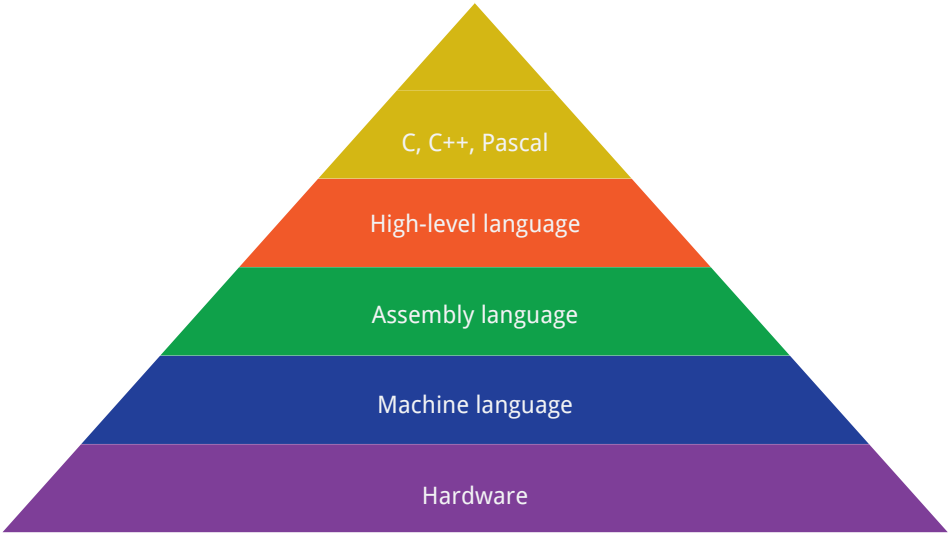


Figure 5.1: Abstraction of computer language

Programmers can use text editors to program in Python. Advanced **integrated development environments** (IDEs) such as PyCharm come with text editors and the shell. An IDE is a software suite that consolidates the basic tools required to write and test software. IDEs increase programmer productivity by introducing features such as editing source code, building executables and debugging.

The integrated development learning environment (IDLE) is the default Python editor that has been available on Raspbian for many generations. It has a built-in interpreter, which allows you to run commands one at a time to test code. However, it does not show line numbers, and it only works with Python. There are two ways to use the interpreter: **interactive mode** and **script mode**. You type Python programs in the interactive mode, and the interpreter prints the results. Alternatively, you may save code in a file and use the interpreter to run the contents of the file, which is referred to as a script. To begin, we will work in an interactive mode.

VOCABULARY

Integrated development environment – *software for building applications that combine common developer tools into a single graphical user interface*

Interactive mode – *back-and-forth engagement between a computer and the user*

Script mode – *used when the user is working with more than one single code or a block of code*

Open IDLE by selecting the Raspberry Pi logo in the top-left corner and click Programming > Python 3 (IDLE). You should be presented with the Python interactive interpreter. To write a program, go to File > New File. Enter in your code.

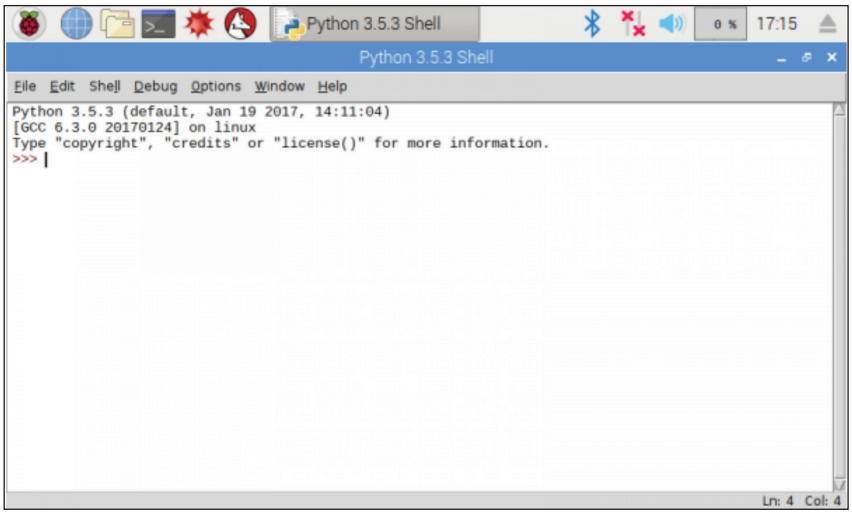


Figure 5.2: IDLE interface

Alternatively, from the Raspberry terminal, type `python3` to activate the IDE. The top Python IDEs are: IDLE, PyCharm, VSCode, Sublime Text3, Atom and many others.

5.1 Introduction to the Python programming language

5.1.1 Difference between *lexis* and *syntax*

As explained in previous modules, *syntax* is the rules and regulations for writing any statement in a programming language such as Python or C#. If a statement follows all the rules, it is syntactically valid. Consider the code snippet below, which is syntactically invalid:

```
if 6>7{print}
```

If you run the above line of code, the following syntax error will be displayed: *Syntax Error: invalid syntax*. This is because the Python interpreter expects a colon following 7 and not braces { }.

If you type in `if 6>7: print` and run the code, you will not get an error display. However, nothing will be printed, which is obvious, as we did not tell the interpreter what to print.

VOCABULARY

Lexis – *collection of words, phrases and idioms used in a programming language*

The *lexical structure* of a program refers to the collection of fundamental principles that control how you build programs in that language. **Lexis** refers to issues regarding the assembly of words that comprise a statement. It is the most basic grammar of the language, defining things such as variable names and which characters are used for comments. Each Python source file comprises a series of characters, just like any other text file. You may think of it as a series of lines, tokens or assertions. These various syntactic

perspectives complement and strengthen one another. Python is extremely strict about program layout, particularly with regard to lines and indentation. If you have been using another programming language, you must pay close attention to what follows.

5.1.2 Difference between an *instruction* and a *comment*

VOCABULARY

Instructions – *accepted by the computer as input; a series of instructions is processed one by one*
Statement – *instruction that the Python interpreter can execute*
Comment – *line in the code that is ignored by the interpreter during the execution of the program*

An **instruction** is a **statement** that a Python interpreter can execute. For example, *number1 = 10* is an assignment statement, where number1 is a variable name and 10 is its value. An *assignment* is a simple statement that assigns a value to a variable. An assignment in Python, unlike other languages, is a statement and can therefore never be part of an expression.

There are other kinds of statements, such as `if` statements, `for` statements, `while` statements, etc. You will learn more about them later. As discussed in Module 3, statements are also used for declaring variables. `Print` will display the output in the IDE shell section. The Python print statement is often used to indicate output variables. Example 5.1 demonstrates the use of print statements. The code entered after `>>>` is the *prompt* of the interactive Python interpreter. It indicates that the interpreter is ready for Python statements to be typed in. Statement 1 is a **comment** and will not be printed back after pressing the enter key. `Comments` are the lines that are ignored by the interpreter during the execution of the program. (Comments will be discussed in more detail later in this module.)

EXAMPLE 5.1

```
pi@raspberrypi:~ $ python
Python 2.7.16 (default, Oct 10 2019, 22:02:15)
[GCC 8.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> #statement 1
... print('Hello NCV Level 2 students')
Hello NCV Level 2 students
>>> #statement=> nothing will be printed here after variable declaration
... number1=20
>>> #statement 3
... print(number1)
20
>>> "Hello"*3
'HelloHelloHello'
```

Multiple statements can be placed on a single line and separated by semicolons, as indicated below:

EXAMPLE 5.2

```
pi@raspberrypi:~ $ python
Python 2.7.16 (default, Oct 10 2019, 22:02:15)
[GCC 8.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> number1 =45; number2 =89
>>> print(number1 +number2)
134
>>> message="Hello world"; print(message)
Hello world
```

NOTE

You can change the display colours of your IDLE by selecting:
File | Edit | Preferences | Style

Apply the background or foreground colour of your choice.

VOCABULARY

Token – *basic component of source code*

A Python statement ends with the **token** `NEWLINE` character. This means each line in the script is a statement. We can extend the statement over multiple lines using the line continuation character (\). This \ operator is also known as an *explicit line break* or an *explicit continuation*. Example 5.3 illustrates how this operator can be used to separate multiple statements.

EXAMPLE 5.3

```
pi@raspberrypi:~ $ python
Python 2.7.16 (default, Oct 10 2019, 22:02:15)
[GCC 8.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> number1=34; number2=49; number3=34
>>> addition=number1 + number2 +\
... +number3; print(addition)
117

#Output is 117.
```

We can also use parentheses – double brackets, e.g. ( ), { } or [ ] – to write a multiline statement. A line continuation statement can be added inside it. Whatever we add inside a parenthesis will be treated as a single statement, even if it is placed on multiple lines. We can use square brackets [ ] to create a list. If required, each list item can be placed on a single line for better readability. See Example 5.4.

EXAMPLE 5.4

```
pi@raspberrypi:~ $ python3
Python 3.7.3 (default, Jan 22 2021, 20:04:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> stud_names=['Wandile', 'Mogamat', 'Martin', 'Zandile']
>>> print(stud_names)
['Wandile', 'Mogamat', 'Martin', 'Zandile']
>>>

#Output
['Wandile', 'Mogamat', 'Martin', 'Zandile']
```

To combine both text and a variable, Python uses the + character, as illustrated here:

EXAMPLE 5.5

```
pi@raspberrypi:~ $ python3
Python 3.7.3 (default, Jan 22 2021, 20:04:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> first_name='Luvuyo'
>>> print('Hello ' + first_name)
Hello Luvuyo
```

However, the plus symbol is used to combine strings (data values that are made up of ordered sequences of characters). If the other value is a number, the following error will be displayed:

```
>>> print("Hello " + 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Figure 5.3: Python errors

The error is caused by different data types. Data types will be discussed in detail in the following module. For now, replace the + sign with a comma, as shown below, to resolve the error:

```
>>> print("Hello", 5)
Hello 5
```

You can also write it as follows:

```
print(f"Hello {5}")
```

We can use *f-strings* to convert the integer 5 directly into a string using Python internals. The preceding sentence allows the string to be joined to a number.

Literals

Literals, also known as *literal constants*, are fixed values and include integers (e.g. 42), floating-point numbers (e.g. 6.23) and strings (e.g. "Hello World!"). Note that strings need to be between single quotes (' ') or double quotes (" "), as shown in the figure below (example of PyCharm code editor). The PyCharm window is explained in detail in section 5.1.9.

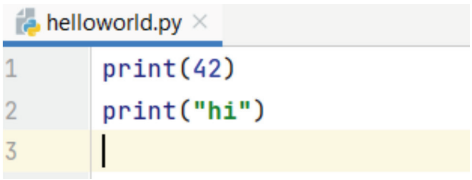


Figure 5.4: Python literals

Variables

As you learned in Module 3, variables are containers of which the value can change. We can store a number or string in a variable and then retrieve that value later.

Good variable names in Python

Follow the guidelines set out here when naming variables.

- Choose a meaningful name instead of short name. Roll_no is better than rn.
- Maintain the length of a variable name. Roll_no_of_a_student is too long.
- Be consistent with your casing; roll_no or RollNo. Read the Python recommended style guide for more information.
- Begin a variable name with an underscore character (_) for a special case.
- Also remember that variable names are case-sensitive, e.g. `X=1`; `x=5`. The two variables will be treated differently.

Example 5.6 shows examples of variables with assigned values.

EXAMPLE 5.6

```
pi@raspberrypi:~ $ python3
Python 3.7.3 (default, Jan 22 2021, 20:04:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> x=1; X=5;
>>> print(x); print(X)
1
5
>>> first_name="Linda"; print(first_name)
Linda
>>>
```

After running the code in the command line interpreter, the variables *x*, *X* and *first_name* will be assigned a value of *5*, *20 and Linda*. If you declare a variable without assigning a value and you press enter, you will get the following error:

```
>>> number1
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'number1' is not defined
>>>
```

The interpreter is reporting an error. While Python is dynamic, which means you do not need to declare a variable until you are going to assign a value to it or use it, it is possible to define a variable without a value, as shown below. You will not get an error.

NOTE

```
>>> number1=int
>>> number2=None
>>> print(number1); print(number2)
<class 'int'>
None
>>>
```

The first variable output reflects a class integer type and the second indicates that there is no value stored in the variable.

Assign value multiple variables

Python allows you to assign values to multiple variables in one line, as reflected in Example 5.7.

EXAMPLE 5.7

```
pi@raspberrypi:~ $ python3
Python 3.7.3 (default, Jan 22 2021, 20:04:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> fruit1, fruit2, fruit3='Mangoes', 'Strawberry', 'Pineapple'
>>> print(fruit1); print(fruit2);print(fruit3)
Mangoes
Strawberry
Pineapple
>>>
```

Assign the same value to multiple variables on one line.

EXAMPLE 5.8

```
>>> fruit1= fruit2= fruit3='Mangoes'
>>> print(fruit1); print(fruit2); print(fruit3)
Mangoes
Mangoes
Mangoes
```

Run the code above and you will see that each variable has a value 'Mangoes'.

As illustrated in Example 5.9, this code snippet adds two numbers. Revise the design techniques you learned in Module 2. Make use of flow charts or the IPO techniques to simplify problem solving.

Planning

For this example, we are going to use an IPO table and flow chart to depict how the problem is solved using design structures.

IPO table

INPUT	PROCESSING	OUTPUT

Enter number_1
Enter number_2
	#Add num1, num2 and store result in answer
	Answer = number_1 + number_2
		Display answer

Flow chart

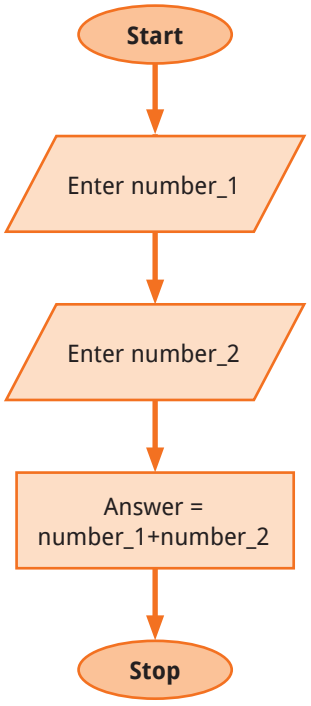


Figure 5.5: Flow chart to depict addition of two numbers

EXAMPLE 5.9

```
>>> #declare first number and assign a value
... number_1=21
>>> #declare second number and assign a value
... number_2=11
>>> #Perform calculation and store sum in a variable called answer
... answer=number_1 +number_2
>>> #Display output
... print(number_1, '+', number_2, '=', answer)
21 + 11 = 32
>>>
```

The last line of our code could be written as below:

```
>>>#Display output
>>>print(str[number_1) + ' + ' + str[number_2) + ' = '
+ str(answer))
```

To consternate (link or join) the code, we substituted the comma (,) with a plus symbol (+). Complete the following tasks to test your understanding.

Task 5.1

Write a Python program using an interpreter to calculate the area of a square. The program uses the side that is predefined. Remember, the formula is:

Area of square = side × side

Task 5.2

Write a Python program that accepts the radius of a circle during development and computes the area. Remember, the area of a circle is computed using the formula pi * r^2.

5.1.3 Defining an *interpreter*

As you learned in Module 3, an *interpreter* is a program that translates a programming language into a comprehensible language. It only interprets one program statement at a time. Interpreters are often smaller than compilers. Python code written in *.py files* is first compiled into intermediate code, which is then saved in *.pyc* or *.pyo* format. These instructions are written in a low-level format that an interpreter can execute. The Python interpreter is usually installed in the */usr/local/bin/python3.10* directory on your machine. Rather than being performed on the CPU, intermediate instructions are executed on a virtual machine.

The fact that interpreted languages are platform-independent is a popular benefit. Python bytecode may be run on any platform as long as the Python bytecode and the virtual machine are both the same version (Windows, MacOS, etc.).

```
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 5.6: Interpreter

To activate the Python interpreter, type python or *python3* from the terminal on Raspberry Pi and press Enter. The result is shown in the figure below:

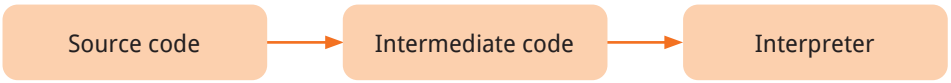


Figure 5.7: Python interpreter

The interpreter prints a prompt (>>>) and waits for you to type in a Python code. Alternatively, activate the interpreter when working with PyCharm IDE.

5.1.4 Comparison between *interpreter* and *compiler*

The differences between a compiler (discussed in the previous module) and an interpreter can be summarised as follows:

INTERPRETER	COMPILER
Translates program one statement at a time	Scans the entire program and translates it into machine code as a whole
Usually spends less time analysing the source code	Analysing the source code takes a lot of time
Slower overall execution time compared to compilers	Faster overall execution time compared to interpreters
No object code is generated, hence memory is efficient	Generates object code that further requires linking, hence requires more memory
Displays errors in each line one by one	Displays all errors after compilation
Based on interpretation method	Based on translation linking-loading model
Does not generate an output program, so the source program is evaluated every time during execution	Generates output program (in the form of *exe*) that can be run independently from the original program
Best suited for the program and development environment	Best suited for the production environment

5.1.5 Python's traditional runtime execution model

Essentially, all the complexity of running Python code is purposefully concealed from programmers. We write code into text files and then execute them through the interpreter. But there is more to it. A fundamental grasp of Python's **runtime** structure can help you to better understand the program execution.

VOCABULARY

Runtime – *final stage of the program lifecycle, when the machine executes the program code*

Bytecode – *low-level, platform-independent representation of the source (Python) code*

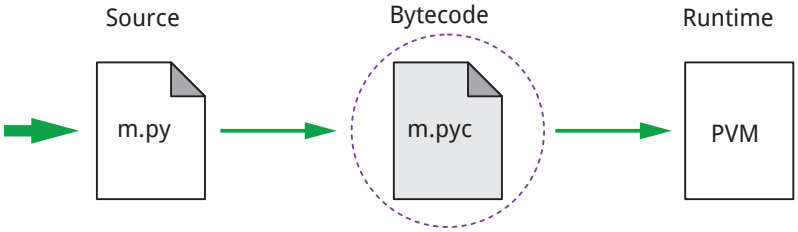


Figure 5.8: Python's traditional runtime execution mode

Python scripts (source code) are first compiled into bytecode before being sent to a Python virtual machine for execution. Python will save the **bytecode** with the extension *.pyc* (which stands for *compiled.py* source).

When you run the source code again, Python will load the bytecode and skip the compilation stage unless you have changed the code. Your code is automatically compiled and then interpreted.

5.1.6 Reasons for using the .py extension

A *.py* file contains source code produced in the Python computer language. To run a *.py* script, use the interpreter included with the Python programming environment. To build or update a *.py* script, use a code editor (such as Sublime) or an IDE (such as PyCharm). A *.py* file contains the source code of a program, whereas a *.pyc* file contains the bytecode of the program, as explained above.

5.1.7 Writing the code to display "Hello World" as output

To test a short amount of code in Python, sometimes it is quickest and easiest not to write code in a file. This is made possible because Python can be run on the command line itself. Type the word *python* or *python3* on the Raspberry Pi terminal. From there you can write any Python statements (code), including "Hello World", as shown in the figure below.

```
Type "help", "copyright", "credits" or "license" for more
information.
>>> print("Hello World")
Hello World
>>>
```

Figure 5.9: Sample project on interpreter

As soon as you press the Enter key, the text "Hello World" is printed back. When you are done using the Python command line, you can simply type `quit( )` or `exit( )` to quit the Python command line interface.

Any text editor can be used to write Python code and run it from the Raspberry pi terminal. If you have a file called *helloworld.py* on your desktop, you will be able to run the following command:

EXAMPLE 5.10

```
pi@rasberrypi:~/Desktop $ python3 helloworld.py
The output will be displayed on the terminal
```

5.1.8 Creating a new project using the high-level IDE

Introduction to PyCharm

Some Python IDEs are preinstalled on Raspberry Pi computers. These include Thonny and BlueJ. As these IDEs are not particularly resource-intensive, you can use them on any Raspberry Pi model you own to create programs and applications in different languages. However, they are not very powerful. Many factors contribute to making PyCharm the most comprehensive and

complete IDE for the Python programming language. PyCharm can also be customised according to production needs and personal preferences. The PyCharm IDE includes code analysis tools, a debugger, testing tools as well as version control options.

Common features of PyCharm IDE

The following are features of the PyCharm IDE:

- Project and code navigation – developers can easily switch between classes, files and methods
- Intelligent code editor – allows high-quality Python code to be written
- Multi-technology development – Python IDE supports common web technology, such as CSS, TypeScript, HTML, CoffeeScript, JavaScript and others
- Availability of integration tools – PyCharm promotes the incorporation of a number of instruments to help boost the productivity of code.

The Raspberry Pi does not come preinstalled with PyCharm. You need to install PyCharm.

HOW TO INSTALL PYCHARM ON RASPBERRY PI

1. Go to *https://www.jetbrains.com/pycharm/download* and select the *community edition*.
2. Make sure the operating system selected is Linux.
3. Navigate to the Downloads page on the Raspberry Pi and you will see your download file.
4. In Downloads, right-click the file and select *Extract here*.
5. Once extraction is done, navigate to the bin inside the extracted file and select *Bin*.
6. Select *Pycharm.sh* and click *Execute*.

REMEMBER

Before doing anything else, make sure Java is installed. If you try to run PyCharm directly on a fresh Raspberry Pi OS installation, it will not work. You can run the following command from the Raspberry terminal:

sudo apt install openjdk-11-jdk

7. PyCharm will now open on your desktop.
8. Select Configure at the bottom of the page and set up a desktop shortcut.
9. If you miss this, you will have to set up the shortcut manually.

How to create the shortcut of PyCharm on the Raspberry Pi menu

Follow the steps set out below to create the PyCharm shortcut:

- Go to the Raspberry Pi menu, select Preferences and click on Main Menu Editor (refer to the figure on the next page).

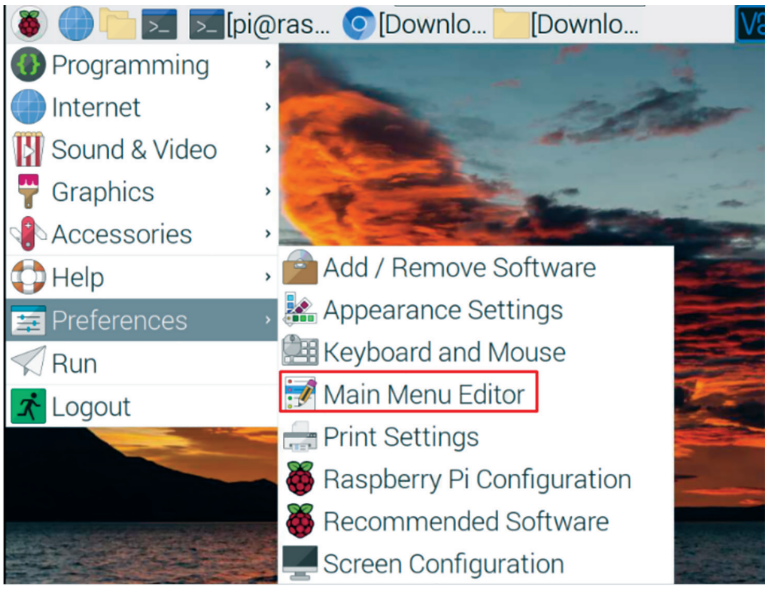


Figure 5.10: Pycharm installation on Raspberry Pi

- Click New Item on the right-hand side of the window.
- A dialogue box (like the one shown below) will appear. Type in the following (as illustrated below):
  - Name: `PyCharm`
  - Command: `/opt/pycharm-community-2021.3.3/bin/pycharm.sh/` (This is the path address where the executable *pycharm.sh* file is available.)
  - Picture: The *png* logo of PyCharm is available in the `/opt/PyCharm-community-2021.3.3/bin/`
- After this, click the OK button in the Main Menu Editor.

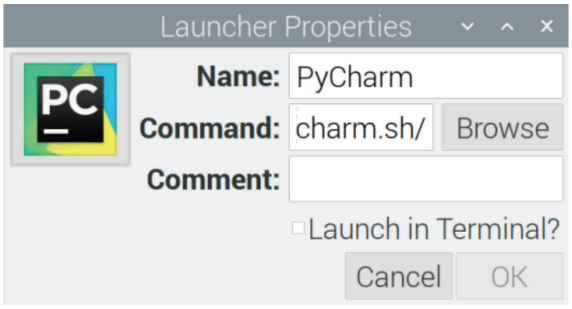


Figure 5.11: Creating the shortcut for PyCharm

- Your program is now on the list of programming applications.
- The application PyCharm will be launched.

The PyCharm window

The figure below shows the default PyCharm interface when you open a project.

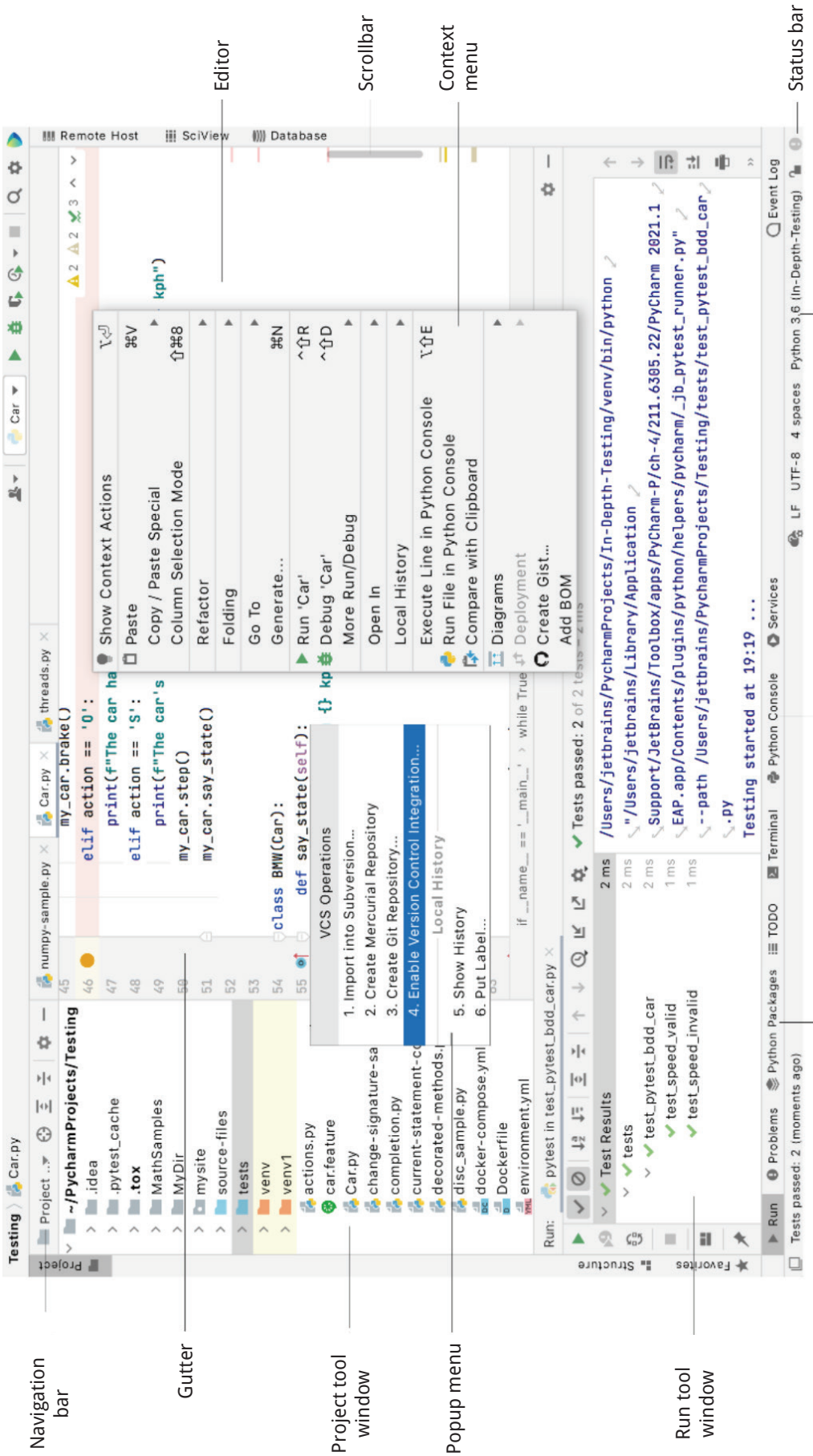


Figure 5.12: PyCharm screen

Configuring user interface themes

Interface themes define how windows, dialog boxes, buttons and other visual elements of the user interface appear. PyCharm uses the Darcula theme by default, unless you change it during the first run. You can change the theme by navigating to:

File → Setting → Appearance and Behaviour. Select your choice and click OK.

Configuring PyCharm settings

PyCharm allows you to configure settings on two levels: the project level and the global level. Settings applied globally apply to all projects opened with a specific installation or version of PyCharm. The settings applied at the project level affect the current project being worked on by the user. To configure your IDE, select File | Settings and select the setting which you want to configure. Alternatively, press Ctrl + Alt + S.

[Opens a general page?]

eLINK

~~For more information on~~ configuring the PyCharm environment~~, visit~~:
~~https://www.jetbrains.com/help/pycharm/quick-start-guide.html~~

5.1.9 Writing the code to display "Hello World" using PyCharm

You are now ready to create your first project using PyCharm IDE. Follow the steps set out here:

- Start PyCharm, select New Project and name the folder in which you will save your project *NCVL2_Python*.
- Python best practice is to create a virtual environment (*virtualenv*) for each project, as Python is not great at dependency management. However, in this section, we are not going to use a virtual environment.
- You may also select a previously configured interpreter. This allows you to choose from among the following environments:
  - *Virtualenv Environment*
  - *Conda Environment*
  - *System Interpreter*
  - *Pipenv Environment*
  - *Poetry Environment*.
- You can choose *Pipenv Environment* and select *Base Interpreter* or select the path for *python.exe*.
- Right-click your folder name and select new python file.
- Also, deselect the Create a main.py welcome script checkbox, because you will create a new Python file for this tutorial.
- Name the file *helloworld* and click Enter. By default, the file is assigned a *.py* extension.
- Type in the code.

EXAMPLE 5.11

```
# This program will print the hello world on the console

"""

print is used to display output on to the screen
statements in the ( ) are treated as a single statement

"""

print("Hello World")
```

Run your application

Use either of the following ways to run your code:

- Right-click the editor and select Run 'Helloworld' from the context menu; or
- Press Ctrl + Shift + F10.

You will see the output on the Python console output window.

- Here you can enter the expected values and preview the script output.
- Note that PyCharm has created a temporary *run/debug* configuration for the **helloworld** file.

Debugging

As you know, programming errors are called *bugs* and the process of tracking them down is called *debugging*. To find the exact place where you made a programming mistake, you need to debug your code using a debugging tool such as PyCharm. You then know which corrections you need to make to the code. Debugging tools often let you make temporary changes so that you can continue to run the program.

Different kinds of errors can occur in a program. It is useful to distinguish among them in order to track them down more quickly:

1 Syntax errors – errors produced during the translation process

A syntax error usually indicates that something has gone wrong with the grammar rules of the programming. An example of a syntax errors is omitting the bracket in the print statement, which will result in an error. In addition, the IDE is extremely smart, as it can tell you if there is an error in the code before you run it. PyCharm underlines the file name in red. Consider the following statement in PyCharm IDE and the error message in the message window:

```
#adding two numbers
answer=5+6+

Output
answer=5+6+
^
SyntaxError: invalid syntax
```

2 **Runtime errors – produced by the runtime system (i.e. if something goes wrong while the program is running)**

Runtime error messages typically contain information about where the error occurred and what functions were being executed. For example: An infinite recursion leads to a runtime error of recursion depth exceeded. The code snippet below shows an example of a runtime error. The user was supposed to input a number, but instead typed the letter 'h', which is a string. The interpreter reports an error as it tries to convert it to an integer.

```
>>> first_number=int(input("Enter the first number"))
Enter the first number h
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'h'
>>>
```

3 **Semantic errors – problems with a program that compiles and runs but does not do the right thing**

These are also referred to as *logic errors*. In some cases, an expression will be evaluated in an unexpected order, resulting in a surprising result. Let us consider the following evaluation aimed at finding the average of two numbers:

```
Average=7+8/2
```

This will result in a logical error. Look at the code snippet below and see how the two statements produce different results:

```
>>> average=7+8/2 #incorrect statement
>>> print(average)
11.0
>>> average=(7+8)/2 #correct statement
>>> print(average)
7.5
>>>
```

User input

By using the input function, you can ask a user to enter information into the terminal. This will prompt the user to type some text, including numbers, before pressing Enter to submit the text. When the user submits, the input method will read the content and return it as a string that may be kept in a variable. Whatever appears within the parentheses (referred to as *arguments*) will be displayed on the screen before receiving user input.

Functions are code parts that may be invoked by name. `Print( )`, for example, is a function that accepts parameters and prints them to the terminal. In the example below, a comma is used to separate two distinct parameters in `print( )`. In this scenario, `print( )` will output the several strings or variables on one line in sequence.

Note that you can use the `int( )` function to turn a string into an integer, assuming the string is an integer. This process of forcing one data type into another is called *typecasting*. We are going to discuss the concept of data types

in the next chapter. The example below illustrates how users can supply input through the keyboard.

EXAMPLE 5.12

```
#Declaring a variable called message and prompting user for
input
message = input("Type your greeting message when
prompted:")
print("You said:", message)
#Declaring a variable called number_1 and prompting user
for input
number_1 = int(input("Type a number:"))
print("You entered:", number_1)
```

Output

```
Type your greeting message when prompted: How was your
weekend colleagues. Mine was great.
You said: How was your weekend colleagues. Mine was
great.
Type a number:5
You entered: 5
```

Let us run the additional programs together to better understand how Python may be used to solve mathematical issues.

Task 5.3

Write a program to accept two numbers from the keyboard during runtime and display the sum.

Concatenating strings

Remember, *concatenating* means joined or linked. Concatenating values stored in values entered by the user at runtime do not differ from how you would enter the values if the input were entered during coding. Example 5.13 challenges you to join values stored in two or more variables. Try it.

EXAMPLE 5:13

Write a program that asks for the user's first name and last name (two separate input ( ) calls)) and then prints the user's first and last name on one line. An example of this program running should look like this:

```
Enter your firstname: Robert
Enter your surname: Marley
My fullname is Robert Marley
```

In some instances, your result might have a decimal point and the client might request that you format it in a certain way. For example, monetary calculations must be formatted correctly to two decimal places. In Python, to print two

decimal places, we will use *str.format( )* with {*:.2f*} as a string and float as a number. Call print and it will print the float with two decimal places.

Look at the following example. The unit price of an item is given as R13,66. What is the total price for all six items? Here you can see how to format the output to a specific number of decimal places.

Formula

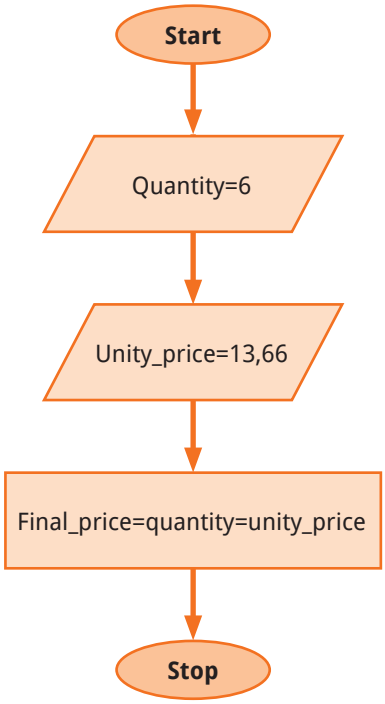
Total_price = quantity × unity_price

Planning

INPUT	PROCESSING	OUTPUT

Assign quantity=6
Assign unity_price=13.66
	#Calculation
	final_answer=quantity*unity_price
		Display total_price

Flow chart



EXAMPLE 5.14

```
#Assign value for the unit_price
unity_price=13.66
#Assign value for the quantity
quantity =6
#Calculation
total_price=quantity * unity_price
#formatting output
final_answer="{:.2f}".format(total_price)
print(final_answer);
```

Output

```
81.96
```

In the above example, the program could be made shorter by applying formatting in the same line as the calculation, as shown below:

EXAMPLE 5.15

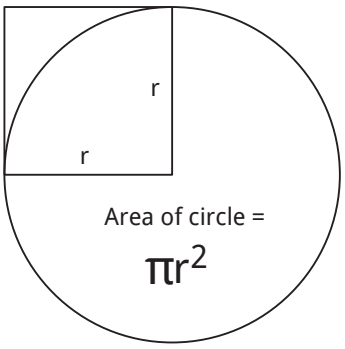
```
#Calculation & formatting output
total_price="{:.2f}".format(quantity * unity_price)

#Displaying output
print(total_price);
```

Task 5.4

Write a Python program that accepts the radius of a circle from the user and computes the area. Make sure the output is formatted correctly to two decimal places.

Python: Area of a circle

In geometry, the area enclosed by a circle of radius r is πr^2. Here the Greek letter π represents a constant, approximately equal to 3,14159, which is equal to the ratio of the circumference of any circle to its diameter.



Task 5.5

Reginald went to a shop to buy a few LEDs to use with Raspberry Pi computers. The number bought and the price of the LEDs are entered by the user through the keyboard. The amount due must include 15% VAT. Write a program to calculate and display the total number of the LEDs.

5.1.10 Comments in Python

Python comments

Comments are for developers. They describe parts of the code where necessary to facilitate the understanding of programmers, including yourself. Comments can be used to:

- explain Python code;
- make the code more readable; and
- prevent execution when testing code.

To write a comment in Python, simply put the hash symbol # in front of the desired comment:

EXAMPLE 5.16

```
>>># This is a python comment
>>>print("Python comments")

Output
/Users/Admin/future/test.py"
Python comments
Process finished with exit code 0
```

Looking at the above example, you will see that the statement that follows the # sign is not displayed as output. Comments can be placed at the end of a line. Python will ignore the rest of the line:

```
Print("Hello World!") #This is a comment
```

If we end the statement above with a # sign, nothing will be printed in the output window.

Multiline comments

Python does not really have a syntax for multiline comments. To add a multiline comment, you could insert a # for each line. The # is called an *octothorpe*.

```
>>> #This is a a comment
>>> #written in more than
>>> #one line
>>> print("Helloworld")
Helloworld
>>>
```

Figure 5.13: Multiline comments

Alternatively, you can use a multiline string. As Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code and place your comment inside it. Here is a syntax to show you how you can use multiple comments in Python:

EXAMPLE 5.17

```
>>>"""
  I can use these quotes to
  act as multiline comments
  and this works
"""
```

5.1.11 Block-structured language

Block-structured languages are high-level languages in which a program is made up of blocks. A block consists of a sequence of statements and/or blocks, preceded by declarations of variables. A block of code may be the body of a

subroutine or function or it may be controlled by a conditional execution (*if* statement) or repeated execution (*while* statement, *for* statement, etc.). Python employs indentation to represent a program's block structure. Below is an example of block structure:

```
X=7;Y=10;
if X > Y:
  print (Y)
  print(X)
print("Hello all")
print("The end")
```

Execution of statements in the print function is determined by the block in which it lies. In the above example: print (Y) and print (X) are controlled by the *if* statement block. Run the program as it is and check your answer. Change the sign and compare the output.

A Python code block tells you which pieces of code are part of a loop, class or function. (Revise the block code discussed in Module 3.)

5.1.12 Using whitespace indentation for block code in Python

All coding rules for the Python code comprising the standard library in the main Python distribution are found in the PEP 8 document. PEP stands for *Python enhancement proposal*.

Python indentation

Python is meant to be an easily readable language. Its formatting is visually uncluttered, and it often uses English keywords where other languages use punctuation. Unlike many other languages, it does not use curly brackets to delimit blocks, and semicolons after statements are optional. It has fewer syntactic exceptions and special cases than C or Pascal.

Indentation refers to the spaces at the beginning of the code line. Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important. Python uses indentation to indicate a block of code. A block of statements can be executed from top to bottom unless a *jump/go to statement* is used.

Unlike other languages such as JavaScript, Python does not employ brackets or begin/end delimiters to mark blocks of statements; instead, indentation is used. The whitespace at the left of each logical line in a Python program indents it. The block is terminated by a logical line with less indentation. All statements in a block, as well as all clauses in a compound statement, must have the same indentation. The standard Python indentation level is four spaces. In a source file, the initial statement must be indented (i.e. it must not begin with any whitespace). In addition, statements typed at the interactive interpreter prompt >>> must have no indentation. Limit all lines to a maximum of 79 characters.

The code block should look as follows:

```
X=7;Y=10;
if X > Y:
  print (Y)
  print(X)
print("Hello all")
print("The end")
```

In the example above, the last two statements would be printed out. Discuss with your classmates:

(a) Why is this so?

(b) What would be printed if you move back line 4 (`print("Hello all")`)?

For flowing, long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters.

Forgetting to indent

```
X=7;Y=10;
if X > Y:
  print (Y)
  print(X)
print("Hello all")
print("The end")

Output
print (Y)
^

IndentationError: expected an indented block after 'if'
statement on line 2
Process finished with exit code 1
```

If you forget to indent the code after the semicolon, as in the example above, the Python interpreter will indicate an *IndentationError error*.

5.1.13 Whitespace indentation in Python

Python is well known for its unusual syntax: Rather than being separated by curly braces or begin/end keywords, blocks are delimited by indentation (or whitespace). Braces make the programmer's job significantly more complicated. Modern IDEs assist, but occasionally, especially when copy-pasting code snippets, things fall out of sync: We fail to get all the closing braces for a block or we place the block at a different level of the hierarchy. This necessitates different forms of indentation. The IDE can occasionally figure out what we are attempting to achieve, but not always. Then we must dig around to ensure that all of the closing brackets are in the correct locations and that the indentation is accurate. Half of the difficulty is avoided by using a whitespace language: You must just get the indentation correct.

The number of spaces is up to you as a programmer. The most common use is four, but it has to be at least one.

Usage of whitespace

Avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets or braces:

```
# Wrong:
Spam( ham[ 1 ], { eggs: 2 } )
```

- Between a trailing comma and a following close parenthesis:

```
# Wrong:
print (x, )
```

- Immediately before a comma, semicolon or colon:

```
# Wrong:
if x == 4 : print(x , y) ; x , y = y , x
```

- More than one space around an assignment (or other) operator to align it with another:

```
# Wrong:
num1 = 1
num2 = 2
num3 = 3
```

OPENS ENTIRE PAGE, not a specific site.

eLINK

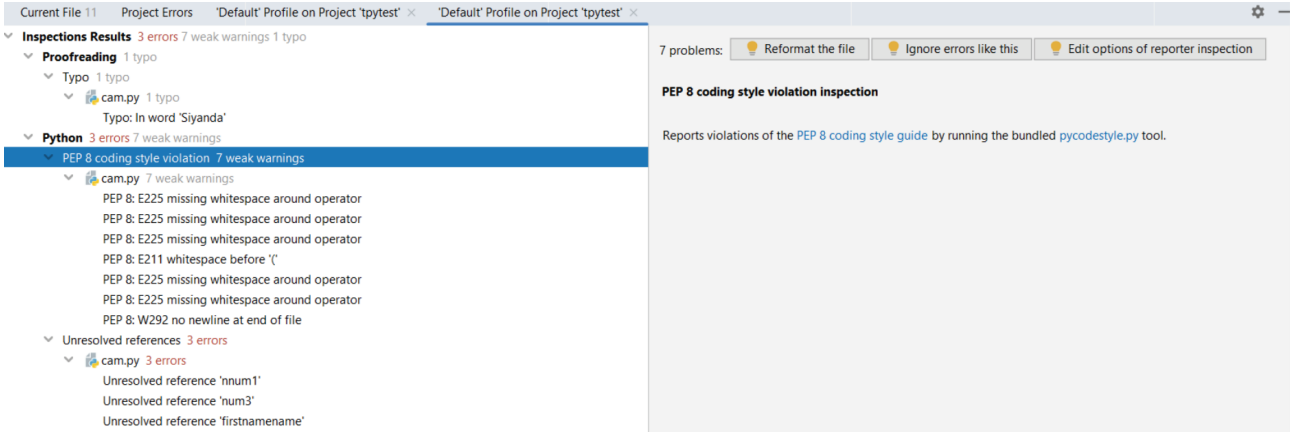
~~*For more information on configuring the PyCharm environment, visit:*~~
~~https://peps.python.org/pep-0008/#whitespace-in-expressions-and-statements~~

5.1.14 IDE message window

Using advanced IDEs such as PyCharm makes development easy for programmers. Error tracing is enhanced by these platforms. PyCharm analyses the code from the modified files by running inspections from the selected profile. If any errors or warnings are detected, you will see a notification.

You can analyse the source code easily by going to the menu bar and selecting: Code → Inspect Code.
Run Analyze | Inspect Code and specify *Whole project* as the scope of analysis. Look at the code snippet below.

The following report details where the errors are in the message window:



So, you can then select each error and get the finer details regarding where to make corrections.

Activity 5.1 **INDIVIDUAL ACTIVITY**

1. What is a *high-level programming* language? (2)
2. IDLE is the abbreviation for .... (1)
3. Explain the difference between *syntax* and *lexis*. (4)
4. Define the term *instruction* as applied in Python programming. (2)
5. With the aid of examples, list TWO ways to implement multiline statements in Python. (3 × 2) (6)
6. True or false: Python is case-sensitive when dealing with identifiers. (1)
7. Which of the following is the correct extension of the Python file? (1)
   - **A** .python
   - **B** .pl
   - **C** .py
   - **D** .p
8. Which of the following is used to define a block of code in Python language? (1)
   - **A** Indentation
   - **B** Key
   - **C** Brackets
   - **D** All of the above
9. What is *PEP 8*? (2)
10. Use the concept of declaring a multiple statement in one line. Declare the first_name and age, respectively, and assign the following values in one line: "Amanda", 23.
    Also print the output in one line. You can use the Raspberry Pi interpreter or PyCharm. The output should be:
    Amanda is 23 years old (5)
11. Look at the example given below and explain why you would get an error message after running the program. (2)

```
num_1 = 7 # first number declaration
num_2 = 8 # second number declaration
answer = num_1 + num_2 # result

#displaying the answer
print("The answer is" + answer)

Output
Print("The answer is " + answer)
TypeError: can only concatenate str (not "int")
to str
```

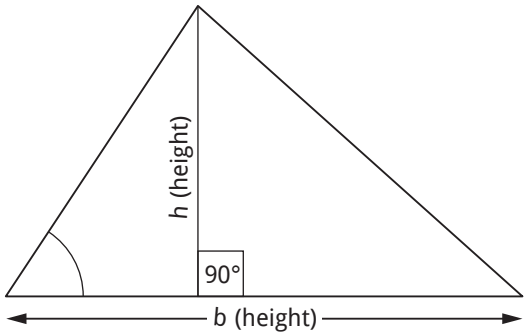
    Rewrite the code to correct the error.
12. List FOUR rules for naming variables in Python. (4)
13. Define the term *interpreter* as applied in Python. (2)
14. Define the term *debugging* as applied in programming. (2)

Activity 5.1 (continued)

15. Write a Python program, using PyCharm IDE, that will accept the base and height of a triangle and compute the area.

A triangle is a polygon with three edges and three vertices. It is one of the basic shapes in geometry. A triangle with vertices A, B and C is denoted as triangle ABC.

- Vertex of a triangle: the point at which two sides of a triangle meet
- Altitude of a triangle: the perpendicular segment from a vertex of a triangle to the line containing the opposite side
- Base of a triangle: the side of a triangle to which an altitude is drawn
- Height of a triangle: the length of an altitude



$$\text{Area A} = \frac{1}{2} \times \text{base} \times \text{height}$$

$$A = \frac{1}{2} \times b \times h$$

(10)

17. Write a program that will produce the output shown below:

a string that you "don't" have to escape

```
This
is a ....... multi-line
heredoc string --------> example
```

(5)

TOTAL: 50

Summative assessment

1. What will the output of the following Python statement be? (1)

```
>>>"a"+"bc"
```

A Bc
B abc
C a
D bca

2. Which of the following is an invalid variable? (1)

A my_string_1
B 1st_string
C foo
D _

3. Which of the following is an invalid statement? (1)

A abc = 1,000,000
B a b c = 1000 2000 3000
C a,b,c = 1000, 2000, 3000
D a_b_c = 1,000,000

4. Create a program that asks the user to enter his/her name and age. Print out a message addressed to the user that tells him/her in which year he/she will turn 100 years old. (10)

Note: For this exercise, the expectation is that you explicitly write out the year (and therefore it will be out of date next year).

The output should be as follows:

```
What is your name: Hamunyari
How old are you: 32
Enter current year: 2022
Hamunyari, you will be 100 years old in the
year 2090
```

5. Write a Python program to convert the temperature in degree Celsius to Fahrenheit. (8)

The formula is :

```
°Fahrenheit=°Celsius*9/5+32
```

Where °Celsius is the temperature in degrees Celsius.

The output should look as below:

```
Enter temperature in Centigrade: 21
Temperature in Fahrenheit is: 69,8
```

6. List FOUR common features of PyCharm IDE. (4)

7. Explain the term *block-structured language*. (2)

Summative assessment (continued)

8. Consider the code below to calculate the average of three integer numbers and displaying the output:

```
#Accepting input
number_1=int(input("Enter first number"))
number_2=int(input("Enter second number"))
number_3=int(input("Enter third number"))

#Process
average=number_1+number_2+number_3/3
result="{:.2f}".format(average)
print(f"The average of {number_1}, {number_2},
{number_3}, is {result}")
```

Output:

```
Enter first number5
Enter second number6
Enter third number7
The average of 5, 6, 7, is 13.33
```

The program is giving an incorrect result.

8.1 What type of error has been made? (1)

8.2 Give the correct line of code to correct the error. (2)

8.3 What is the purpose of `"{:.2f}"`.format in the code? (2)

TOTAL: 32

Module 6

Data types, variables and output

After you have completed this module, you should be able to:

- explain generic concepts covered in modules 3 and 5 – *integer*, *float*, *Boolean* and *string data* – within the given contexts;
- define *arithmetic operations*;
- list the different arithmetic operators;
- define *binary operator* and *operand*;
- list and explain (with examples) what a binary arithmetic expression consists of;
- write Python code that uses *print( )* to display the value of different arithmetic expressions;
- list and explain the data produced based on the data type of the operands involved;
- list and explain the rules involved when creating complex arithmetic expressions containing multiple operands;
- list and describe the three levels of precedence;
- write Python code that uses `print( )` to display the value of different complex arithmetic expressions;
- understand the following number systems:
  - decimal system
  - binary system
  - conversions;
- define the term *positional number system*;
- indicate the base of a number using the base subscript;
- define and explain the bases of a decimal number;
- rewrite a decimal number as the sum of its base;
- define and explain the base of a binary number;
- convert a binary number to a decimal number;
- convert a decimal number to a binary number;
- define and explain the base of a hexadecimal number;
- explain how computers use hexadecimal numbers;
- list the hexadecimal numbers 0–1, A–F and their decimal equivalents;
- convert a decimal number to a hexadecimal number;
- list the different types of data associated with characters and strings;
- initialise data as a character;
- initialise data as a string
- manipulate character and string data for output purposes;

- use common built-in functions to manipulate strings and characters;
- use concatenate strings;
- convert a string to another data type;
- explain the purpose of each parameter used in the `open( )` function to ready a file for writing using the *wt* mode; and
- write Python code that:
  - opens a text file for writing
  - will use the file object to write to a text file
  - ensures that a file is successfully opened and displays an appropriate error message if not
  - closes a file.

Introduction

In Module 3 you learned how to apply various programming concepts as part of the coded solution. You also learned how to write simple code in Scratch, a high-level yet easy block-based, visual programming language. Python, another high-level programming language, and the various concepts associated with writing programs were discussed in Module 5. After briefly refreshing your knowledge of these generic concepts, you will learn more about the Python data types and the arithmetic operations and number systems used in programming. You will practise working with characters and strings as well as writing Python codes.

6.1 Revising problem solving concepts mastered with Scratch

By using Scratch, MU and PyCharm IDEs in previous modules, you have learned some operations and basic programming skills. In this section, you will implement the concepts you have learned using block-based programming. We will use Python to turn the LED lights connected to a Raspberry Pi on and off.

Task 6.1

Write a Python program to turn the LED lights on and off, using the GPIO of the Raspberry Pi (revise the relevant section in Module 3).

You will need:

- 3 × LEDs
- 6 × jumper cables
- 1 × breadboard
- 3 × 330-Ω resistors

REMEMBER

You will need to connect the LED lights and the resistors to the breadboard and then link them to the Raspberry Pi using the jumper cables.

Instructions

1. Start PyCharm on your Raspberry Pi and save the project.
2. By now you are expected to know how to connect the LED lights and the resistors to the breadboard. If you need guidelines, refer to Module 3.
3. Connect the LED lights to the GPIO header as follows:
   - Red = GPIO (22)
   - Amber = GPIO (27)
   - Green = GPIO (17)
4. Ensure that the negative jumper cables are connected to one of the ground pins of the Raspberry Pi. (You should know how to make the connection. If you need to refresh your memory, refer to Module 3.)

By now, you understand the meaning of **import** in Python. To use the GPIO header for physical computing, we will need to import the *gpiozero* library.

The program will also need to import a time module so that the lights will turn on after a specific time.

Here is the source code:

```
#Program to turn ON/OFF the lights using Raspberry Pi
from gpiozero import LED
from time import sleep
#Declaring the 3 variables -red, amber and green
red = LED(22)
amber = LED(27)
green = LED(17)
#Turning On/Off the lights using a sequence
red.on( )
sleep(1)
amber.on( )
sleep(1)
green.on( )
sleep(1)
red.off( )
sleep(1)
amber.off( )
sleep(1)
green.off( )
```

Run your program. The lights will turn on and off.

The program can be made shorter by implementing an iteration structure. The challenge will be repeated once you have learned about *iteration structures* using Python (Chapter 9).

VOCABULARY

Import – *to bring a file from a different program into the one you are using*

Sequence – *default control structure; instructions are executed one after another*

Using a sequence structure to implement a buzzer on Raspberry Pi

A buzzer is kind of a bell. You will need to write a program to sound the buzzer at any given time. While using selection structures would be the best choice, you are going to learn how to implement the program using a ***sequence*** *structure*.

You will need:

- Buzzer
- Breadboard
- 330-Ω resistor

Instructions

You should be familiar with connecting externals to the GPIO header. However, the steps are repeated here for the sake of revision.

Step 1: Open the GPIO header (usually covered with rubber to protect the pins).

Step 2: Connect the Raspberry Pi to the ground rail of the breadboard, taking care to align the pins – no pins should be bent or crushed.

Step 3: Connect the pins from the Raspberry Pi to the breadboard using jumper cables.

Step 4: Connect the buzzer to the breadboard. A buzzer has two cables – the longer one is the positive and the shorter one the negative lead. Insert the buzzer with the two component legs on separate rows.

Step 5: Connect the negative lead of the buzzer to the ground rail using a jumper wire.

Step 6: Connect the buzzer to Pin 28 via the resistor. One of the legs of the resistor must be in line with the positive lead on the buzzer. Slot the other leg into any unused breadboard row.

Step 7: Connect the negative leg of the resistor in the same row as the negative leg of the buzzer.

Activity 6.1 **INDIVIDUAL ACTIVITY**

Revise modules 3 and 5 to find the answers to the questions that follow.

1. Name the THREE programming structures. (3)
2. How many pins are on Raspberry Pi 4? (1)
3. Explain the term *global variables* as applied in programming. (2)
4. What is meant by the *runtime* of an error? (2)
5. Distinguish between the terms *operator* and *operand*. (4)

TOTAL: 12

6.2 Python data types

A data type determines the value of an object as well as the operations that can be performed. The following concepts will be covered in this section:

- Operators
- Operator precedence
- Arithmetic expressions.

6.2.1 Data types

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserve memory. Therefore, by assigning different data types to variables, you can store **integers**, **decimals** or **characters** in these variables. In programming, data types are an important concept.

VOCABULARY

Integer – *data type used to represent real numbers that do not have fractional values*
Decimal – *data type that uses the base-10 numbering system*
Character – *single visual object used to represent, numbers or symbols*

The built-in or default Python data types can be divided into various categories, as shown in table below.

CATEGORY	DATA TYPE (SYNTAX)	EXAMPLE
Text or string	str	`firstname = 'Linda'`
Numeric	int	`number_1 = 1`
	float	`amount=2.50`
	complex	`amount = 5j`
Boolean	bool	`is_Late=True`
Sequence	list	`cars=['Toyota', 'Ford', 'Cherry', 'Nissan']`
	tuple	`cars=('Toyota', 'Ford', 'Cherry','Nissan')`
	range	`number = range(5)`
None	NoneType	`cars = None`
Dictionary	dict	`person={'firstname':'Linda', 'surname':'Ntuli'}`
Set	set	`cars={'Toyota', 'Ford', 'Cherry', 'Nissan'}`

Table 6.1: Data types

Python variables do not need an explicit declaration to reserve memory space. The declaration occurs automatically when you assign a value to a variable. However, the programmer can also specify the data type by using the data type functions, e.g.:

```
firstNumber=int(20)
```

In the example above, *firstNumber* is assigned to a value of `int( )` data type.

To check the data type of a value, we use the keyword type before the variable name or the value, as shown below on the Python shell:

```
>>> number1=10
>>> type(number1)
<class 'int'>
>>> surName="Stephens"
>>> type(surName)
<class 'str'>
>>>
```

If you want to check on an IDE such as PyCharm, you would use:

```
print (type(variablename).
```

You do not need to use the print function if you are typing directly into the REPL. If you are using an IDE such as PyCharm or MU, you will need to use the word *print* before the *type(variable)* statements as follows:

```
number1=20
print(type(number1))
```

Having looked at examples of the different data types, let us examine each type in more detail.

Integers

The abbreviation `int` is used for integer. An integer is a whole number (positive or negative) of unlimited length without decimals. In Python 3, there is effectively no limit to how long an integer value can be. Of course, it is constrained by the amount of memory your system has, but beyond that, an integer can be as long as you need it to be, e.g.:

```
>>> print(123123123123123123123122342343123123123123123123123127 + 1)
123123123123123123123122342343123123123123123123123128
```

For improved readability, Python allows the use of underscore for long numbers, e.g. the following number: count = 10000000000 can be coded as follows:

```
count = 10_000_000_000
```

Float

Any number with a decimal point (positive or negative) is a floating-point number. The term *float* means that the decimal point can appear in any position in a number. In general, you can use floats like integers, e.g.:

```
>>> 5.0-4.0
1.0
>>> 2.5+6.7
9.2
>>>
```

If you mix an integer and a float in any arithmetic operation, the result is a float, e.g.:

```
>>> 5+3.4
8.4
```

Complex

Complex numbers are those numerical values that have both real and imaginary parts. You can declare a variable as a complex number and assign a value to it, e.g. `1+3j`:

```
number=1+3j
```

Boolean

As you know, *Boolean* refers to a system of algebraic notation used to represent logical propositions by means of the binary digits 0 and 1. There are only two possible values for the Boolean data type: true (1) or false (0). Boolean data types cannot be used for arithmetic operations. Most of these variables are used in conditionals (*if* statements).

Syntax:

```
>>> isRetired=True
>>> print(isRetired)
True
```

String

A string value is a collection of zero or more characters put in single, double or triple quotes. A string is used to save one or more characters, and it is the most-used data type. This is a declaration of a variable greeting, which is assigned the string "Hello World".

```
>>> greeting="Hello World"
>>> type(greeting)
<class 'str'>
```

String literals may be delimited using either single or double quotes. All the characters between the opening delimiter and the matching closing delimiter are part of the string. To break up a string over more than one line, include a backslash before each **newline**, and the newlines will be ignored.

VOCABULARY

Newline – *character used to represent the end of a line of text and the beginning of a new line; also known as a line break or end-of-line marker*

```
>>> print('a\
... b\
... c')
abc #output
```

Strings cannot be used for arithmetic operations.

Look at the following example and analyse the answer:

```
>>> num1='5'
>>> num2='6'
>>> print(num1+num2)
56
```

You may have expected the answer to be 11. However, because the values 5 and 6 have single quotes, they are interpreted as strings, hence the concatenation.

VOCABULARY

Slicing – *feature that enables accessing parts of sequences such as strings, tuples and lists; can also be used to modify or delete the items of changeable sequences such as lists*

How to access characters in a string

We can access individual characters by using their index and a range of characters by using slicing. Index starts from 0. Trying to access a character out of the index range will give rise to an *IndexError*. The index must be an integer. We cannot use floats or other data types, as this will result in a *TypeError*.

Python allows negative indexing for its sequences. The index of –1 refers to the last item, –2 to the second-last item, etc. By using the **slicing** operator (:), we can access a range of items in a string.

```
>>> myLevel="NCV Level 2"
>>> print(myLevel) #Display /print the whole variable to the
screen
NCV Level 2
>>> type(myLevel) #Display the data type of the string
<class 'str'>
```

```
>>> print(myLevel[2]) #will print the 3rd character L
V
>>> print(myLevel[10]) #will print the 3rd character 2
2
>>> print(myLevel[2:5]) #will slice from third character to
the 5th character
V L
>>> print(myLevel[:]) # Will print the whole string
NCV Level 2
>>> len(myLevel) #Prints the length of the string including
spaces
11
```

eLINK

For more information regarding Python strings, especially combining strings, formatting strings and common Python string methods, visit the official Python site: https://docs.python.org/3.10/library/string.html

List

A *list* (also called an *array*) is an ordered collection of one or more data items, not necessarily of the same type, placed in *square brackets*. The values that make up a list are called its *elements*. It is a tool for storing data, just like a variable. It is the most versatile data type available. (Refer to Python lists below for more details.)

Python lists

A list contains items separated by commas and enclosed in square brackets – [ ]. Creating a list is as simple as putting different comma-separated values between square brackets.

Elements in a list have the following characteristics:

- They maintain their order unless explicitly reordered (e.g. by sorting the list).
- They can be of any type, and types can be mixed.
- They are accessed via numeric (zero-based) indices.

An example of a list is shown below:

```
list = [123,'abcd',10.2,'d']    #can be an array of any data type or single data type.
list1 = ['hello','world']
print(list)    #will output whole list. [123,'abcd',10.2,'d']
print(list[0:2])    #will output first two element of list. [123,'abcd']
print(list1 * 2)    #will gave list1 two times. ['hello','world','hello','world']
print(list + list1)    #will gave concatenation of both the lists.
[123,'abcd',10.2,'d','hello','world']
```

Accessing values in lists

To access values in a list, use the square brackets for slicing along the index or indices to obtain a value available at that index, e.g.:

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
print "list1[0]: ", list1[0]
print "list2[1:5]: ", list2[1:5]
```

When the above code is executed, it produces the following result:

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

Updating lists

You can update single or multiple elements of lists by using the slice on the left-hand side of the assignment operator, and you can add to elements in a list using the `append( )` method.

Tuple

A *tuple* is an ordered, unchangeable collection of data items, not necessarily of the same type, put in *parenthesis*. Tuples are used to store multiple items in a single variable. They are built-in data types in Python, e.g. list, set and dictionary.

Range

Range is another built-in Python data type. It is mainly used with loops. The `range( )` function returns a sequence of numbers, starting from 0 by default, increments by 1 (by default), and stops before a given number, e.g.:

```
x = range(6)
For n in x:
print(n)
```

NoneType

This is a data type of the object when the object does not have a value. You can initiate the NoneType object using the keyword `None` as follows:

```
obj = None
type(obj)
```

The output will be:

```
<type 'NoneType'>
```

If you try to print the value of the NoneType object, it does not print anything on the Python console.

Dictionary

A dictionary is a collection of key-value pairs where each key is associated with a value. A collection of such pairs is enclosed in curly brackets – { }, e.g.:

```
>>> towns={5: 'Claremont', 2: 'Newlands', 8: 'Rondebosch',
1: 'Mowbray'}
>>> towns[5]
'Claremont'
>>>
```

Elements in a dictionary have the following characteristics:

- Every entry has a key and a value.
- Ordering is not guaranteed

- Elements are accessed using key values.
- Values can be of any type (including other dictionaries), and types can be mixed.

Set

Sets are built-in Python data types used to store multiple items in a single variable. A set is created by placing all the items (called *elements*) inside curly brackets { }, separated by commas, or by using the built-in set( ) function. Sets cannot have two items with the same value. It can have any number of items and they can be of different types. However, a set cannot have mutable (changeable) elements such as lists, sets or dictionaries as its elements.

For example:

```
#Set of integers
my_set = {1, 2, 3}
print(my_set)
Output: {1, 2, 3}
#Set of mixed data types
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
Output: {1.0, (1, 2, 3), 'Hello'}
```

eLINK

Here is a link on Python data types:
https://www.w3schools.com/python/python_data types.asp

Activity 6.2 **INDIVIDUAL ACTIVITY**

1. Under which category of data type will you find a *float*? (1)
2. Explain what a *Boolean data type* is. (2)
3. Explain, with the aid of an example, how to check the data type of a value in Python. (2)
4. What is the limit of the Python integer data type in terms of storage? (1)
5. Define the term *string* as applied in Python data types. (2)
6. Write a Python code snippet to print the letter 'c' in the string South Africa. Use your own variable name. (2)
7. What will the output of the following code be? (1)

```
>>> myResidence='Rondebosch'
>>> print[myResidence[2:5]]
```

8. Consider the following code snippet.

```
X='5'
y='7'
answer=x+y
print(answer)
```

What will the output be? Explain your answer. (2)

TOTAL: 13

6.3 Arithmetic operations

Arithmetic operators are used with numeric values to perform common mathematical operations such as addition, subtraction, multiplication and division.

6.3.1 Different arithmetic operators

There are seven arithmetic operators in Python:

- Addition (+)
- Subtraction (–)
- Multiplication (*)
- Division (/)
- Modulus (%)
- Exponentiation (**)
- Floor division (//).

6.3.2 Binary operator

A binary operator is an operator that operates on two operands and manipulates them to return a result. Operators are represented by special characters or by keywords and provide an easy way to compare numerical values or character strings.

```
Operand1 Operator Operand2
```

For example:

```
result = x + y
```

x and *y* are the two operands and the plus sign (+) is the operator.

In the example above, we used the + operator to add together two values. This operator falls under the arithmetic operators. Python operators in general are used to perform operations on values and variables. These are standard symbols used for the purpose of logical and arithmetic operations. In this section, we will look into different types of Python operators.

Python divides the operators into the following categories:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators.

The arithmetic operators are listed above. Now we will take a closer look at the other operators.

Assignment operators

Assignment operators are used to assign values to variables.

Assume the value of *x* is 5.

ASSIGNMENT OPERATOR	EXAMPLE	ANSWER
=	x = 5	
+=	x+ = 5	10
−=	x− = 5	0

You can do the same with the other arithmetic operators.

Comparison operators

Two values can be compared using comparison operators. The following are examples of comparison operators:

- `==` Equal to

```
>>> x=4; y=4;
>>> x==y
True
```

NOTE

If the data types are not the same, the output will be false. For instance:

```
>>> x=4; y='4'
>>> x==y
False
```

In the above example, {x} is an integer data type and {y} is a string data type. Comparing the two gives a false output.

- `!=` Not equal to

  The operator compares the values on either side of it and decides whether they are equal. It gives a true or false output depending on the values entered. For example:

```
>>> x=4; y=4;
>>> x!=y
True
```

  Other common comparison operators include greater than (>), less than (<), greater than or equal to (>=) and less than or equal to (<=).

Logical operators

Logical operators are used to combine conditional statements. The logical operators used in Python are *and*, *or* and *not*.

- **and** – returns *True* if both statements are true and *False* if one condition is not satisfied.

```
>>> age=18
>>> age>16 and age<65
True
>>>
```

- **or** – returns *True* if one of the statements is true.

```
>>> age=18
>>> age<18 or age <65
True
>>>
```

  In the above example, age is not less than 18, but because the second condition is true, the comparison returns true.

- **Not** – reverses the result and returns *False* if the result is true, as shown in the code snippet below:

```
>>> age=18
>>> not(age<16 and age<65)
True
```

Identity operators

Identity operators (*is* and *is not*) are used to check whether the variables or values are actually located on the the same part of the memory location.

- **is** – evaluates to *True* if the operands on either side of the operator point to the same object; otherwise, the output would be *False*.

```
>>> x=8; y=8
>>> x is y
True
```

- **is not** – evaluates to *True* if the variables on either side of the operator are not identical, i.e. they do not point to the same object; should they refer to the same object, the output would be *False*.

```
>>> a=20; b=30
>>> a is not b
True
>>> x is y
True
>>>
```

Membership operators

A membership operator checks whether a value or variable is found in a sequence (string, set, list, etc.). There are two types of membership operators: *in* and *not in*.

- **in** – returns *True* if the value is found in the sequence
- **not in** – returns *True* if the specified value is not present in the sequence.

Example

```
>>> numbers=[4,7,6,5,89,21,34,45]
>>> 5 in numbers
>>> 89 not in numbers
False
```

Bitwise operators

Bitwise operators are used to compare (binary) numbers. These operators act on operands as if they were strings of binary digits. As the name suggests, bitwise operators operate bit by bit. The following operators are used: &, |, ^, ~, << and >>.

6.3.3 Operands

Operators and operands are explained in Module 3. To recap, in computer programming, an *operand* is a term used to describe any object that is capable of being manipulated.

For example, in `"1 + 2"`, `1` and `2` are the operands and the plus symbol is the operator.

6.3.4 Binary arithmetic expression

EXPRESSION	DESCRIPTION	EXAMPLE

Addition operator	As mentioned previously, in Python + is the addition operator. It is used to add 2 or more values.	`>>> x=5; y=7`
		`>>> x+y`
		`12`

Subtraction operator	The minus symbol (–) is the subtraction operator. It is used to subtract the second value from the first value.	`>>> x=5; y=7`
		`>>> x–y`
		`–2`

Multiplication operator	The asterisk symbol (*) is the multiplication operator. It is used to find the product of two values.	`>>> x=5; y=7`
		`>>> x*y`
		`35`
		`>>> xyx`
		`175`

Division operator	The forward slash (/) is the division operator. It is used to find the quotient when the first operand is divided by the second.	`>>> x=5; y=7`
		`>>> x/y`
		`0.7142857142857143`

Modulus operator	This operator returns the remainder when the first operand is divided by the second. The symbol used to get the **modulus** is percentage (%).	`>>> x=12;y=5`
		`>>> x % y`
		`2`

Exponentiation operator	In Python, two asterisks (**) are used as the **exponentiation** operator. It is used to raise the first operand to the power of the second.	`>>> x=4;y=3`
		`>>> x**y`
		`64`

Floor division	The double forward slash (//) is used to conduct the **floor division**. It is used to find the floor of the quotient when the first operand is divided by the second.	`>>> 12//7` `1`

VOCABULARY

Modulus – *also referred to as* modulo*; the remainder from the division of the first argument to the second*

Exponentiation – *operation where a number is multiplied several times by itself*

Floor division – *operation that divides two numbers and rounds the result down to the nearest integer*

NOTE

Addition and multiplication can be used with strings.
*Addition (+) is used to concatenate and * is used to multiply strings.*

eLINK

~~Here is a link for~~ arithmetic operations:
~~https://youtu.be/Z8vmjToqlO8~~

6.3.5 Code used to display the values of different arithmetic expressions

As discussed in Module 5, the addition operator can be used with strings and the result is a concatenation.

```
>>> firstname="Andy"
>>> surname="Murray"
>>> firstname + surname
'AndyMurray'
>>>
```

If you need to leave a space between the two strings in the output of the preceding example, add another empty string or leave one space in the code after *Andy* before the closing quotation.

Instead of the + operator, you can use the assignment operator (+=) to concatenate multiple strings into one, e.g.:

```
>>> s = 'My South Africa'
>>> s += ' Concatenation'
>>> print(s)
My South Africa Concatenation
>>>
```

When a multiplication operator is used, the variable will be repeated several times.

Example

```
>>> firstName='Zindzi'
>>> firstName*3
'ZindziZindziZindzi'
>>>
```

6.3.6 Output data types of different operands

An expression can take many operators and operands.

Example
Write a Python program to evaluate the following expression:
((((6+4)*2)–10)//2)–4*2

```
>>> print(((((6+4)*2)-10)//2)-4*2)
-3
>>>
```

How did we arrive at –3?

Let us break down the evaluation:

```
(6+4) = 10
(10*2) = 20
(20—10) = 10
(10//2) = 5
4*2 = 8
5—8 = —3
```

In general, you can use floats like integers. The division of two integers always returns a float.

```
>>> 20/10
2.0
>>>
```

Similarly, any arithmetic operation that combines an integer and a float yields a float.

6.3.7 Rules involved when creating complex arithmetic expressions

If an expression contains more than one operator, the order in which they are evaluated is determined by precedence rules. In Python, mathematical operators are evaluated in the same order as mathematical operators.

- *Parenthesis* has the highest precedence and can be used to order the evaluation of an expression. `(1+1)**(5-2)` evaluates to 8 while `1+1**5-2` evaluates to 0.
- *Exponentiation* has the next highest precedence. This explains why `1+1**5-2` evaluates to 0.
- *Division* and *multiplication* have the same precedence, i.e. higher than addition and subtraction (which also happen to have the same precedence).
- Operators with the same precedence are evaluated from left to right. This is known as *left associativity*. In the expression `6-3+2`, the subtraction occurs first, yielding 3. Then we add 2 to get 5.

NOTE

Comparison operators and assignment operators do not support associativity, which means that an expression such as `10<20<30` *does not mean* `(10<20)<30` *or* `10<(20<30)`*. Rather, it means* `10<20` *and* `20<30`*.*

6.3.8 Three levels of precedence

Python has built-in support for complex arithmetic expressions.

- The operator precedence goes from highest (parentheses) to lowest (addition and subtraction).
- If you must compare two operators in the same group, the precedence goes from left (highest) to right (lowest). This is called *left-to-right associativity*.
- For the exponents group, the associativity is right to left.

6.3.9 Complex arithmetic expressions

VOCABULARY

Parse – *to analyse an object specifically/read the program code*

The built-in Python `eval( )` method is used to dynamically evaluate expressions from string or compiled-code input. The word *eval* can be seen as an abbreviation for 'evaluation', which is the process of finding output. When you send a string to `eval( )`, it **parses** it, compiles it to bytecode and evaluates it as a Python expression. When you call `eval( )` with a compiled

code object, the function only executes the evaluation phase, which is useful if you run `eval( )` numerous times with the same input.

```
>>> eval("14 + 10")
24
>>>
```

As you can see in the example, the expression is a string. By default, if `eval` is not used, a concatenation is the result, i.e. '14 + 10'. When you call `eval( )` with a string as an argument, the function returns the value that results from evaluating the input string.

Look at the example below using *print* to display the value:

```
>>> print(eval('6+4*2-10//2-4*2'))
1
>>>
```

Activity 6.3 **INDIVIDUAL ACTIVITY**

1. List the SEVEN arithmetic operators. (7)

2. Define the term *binary operator*. (2)

3. What will the value of the following Python expression be?

```
4 + 3 % 5
```

A 7 **B** 2
C 4 **D** 1 (1)

4. What is the order of precedence in Python?

A Exponential, parentheses, multiplication, division, addition, subtraction

B Exponential, parentheses, division, multiplication, addition, subtraction

C Parentheses, exponential, multiplication, division, subtraction, addition

D Parentheses, exponential, multiplication, division, addition, subtraction (1)

5. Which of the following Boolean expressions is NOT logically equivalent to the other three?

A `not(-6<0 or -6>10)`

B `-6>=0 and -6<=10`

C `not(-6<10 or -6==10)`

D `not(-6>10 or -6==10)` (1)

6. Consider the following declaration:

```
>>> myname='this is not a joke. They are winning'
```

Write a Python code snippet to check whether the word *winning1* exists in *myname*.

What will the answer be? (3)

7. Consider the following statement with a comparison operator:

```
>>> 100=='100'
```

Is this statement true or false? Give a reason. (3)

Activity 6.3 (continued)

8. After typing the following snippet in Python shell, the output is *False*.

```
>>> not 5
False
```

Why is this so? (2)

9. Which TWO arithmetic operators can be used on strings? (2)

TOTAL: 22

6.4 Number systems

Number systems are usually mathematical notations for numbers based on digits or symbols. In this section you will learn more about the different number systems used in Python and how to convert numbers of one system to another.

6.4.1 Positional number system

Numbers are used to represent arithmetic values, count or measure quantities. The number system can be classified into two types, namely:

- positional; and
- non-positional.

In this module, we are only going to discuss the positional number system. Examples of positional number systems include the decimal, binary, octal and hexadecimal number system, BCD, etc., as you are about to learn.

Positional number systems are also known as *weighted number systems*. As the name implies, a weight is assigned to each digit according to its position in the number. From left to right, the weights increase by a constant factor equivalent to the base or radix. The digit position n has weight r^n.

Example

$859 = 8 \times 10^2 + 5 \times 10^1 + 9 \times 10^0$

$12{,}58 = 1 \times 10^1 + 2 \times 10^0 + 5 \times 10^{-1} + 8 \times 10^{-2}$

6.4.2 Indicating the base of a number

A **base** is the available numbers in a numbering system. The most-used base is a base-10 or decimal numbers, which are the numbers 0 to 9.

The number (N) in *base* or **radix** can be written as follows:

$$(\mathrm{N})b = dn_{-1}\, dn_{-2} \text{ -- -- -- -- } d_1\, d_0 \,.\, d_{-1}\, d_{-2} \text{ -- -- -- -- } d_{-m}$$

VOCABULARY

Base – *a number base is the number of digits or combination of digits that a system of counting uses to represent numbers*

Radix – *number of unique digits, including the digit zero, used to represent numbers*

The above example represents numbers that will have decimal commas (also called *floating-point numbers*). The numbers that appear after the decimal point will have a negative subscript. In section 6.4.1, we represented the value 12,58 in a similar way. In the example above, d_{n-1} to d_0 is the integer part, followed by a radix point, and d_{-1} to d_{-m} that makes up the fractional part.

d_{n-1} = most significant bit (MSB) d_{-m} = least significant bit (LSB).

The table below illustrates the different positional number systems and their bases.

BASE	NUMBER SYSTEM
2	Binary
8	Octal
10	Decimal
16	Hexadecimal

6.4.3 Decimal number

The decimal number system has the most important role in the development of science and technology. If the base value of a number system is 10, then it is called a decimal number system. This is the weighted (or positional) number representation, where the value of each digit is determined by its position (or weight) in a number. The 10 numbers used in base-10 are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. Each position in the decimal system is 10 times more significant than the previous position. In our everyday calculations, we use numbers in base-10.

6.4.4 Rewriting a decimal number as the sum of its base

The value of a decimal digit is determined by multiplying each digit by the value of the position in which the digit appears.

Example

The number 1 025 is interpreted as:

$1\,025 = 1 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 5 \times 10^0 = 1\,000 + 0 + 20 + 5 = 1\,005$

The rightmost bit 5 is the LSB and the leftmost bit 1 is the MSB.

6.4.5 Binary numbers

The binary number system, in mathematics, is a positional numeral system employing 2 as the base and so requiring only two different symbols for its digits, namely 0 and 1, instead of the usual 10 different symbols needed in the decimal system. We mentioned earlier that a computer stores data using electrical pulses (0s and 1s). The computer does not understand the text that we type. It understands 0s and 1s, and this is called the *binary language*.

The numbers from 0 to 10 are written in binary as follows:

0, 1, 10, 11, 100, 101, 110, 111, 1 000, 1 001, and 1 010.

6.4.6 Converting a binary number to a decimal number

To convert a binary number to a decimal number, write down the binary number and list the powers of 2 from right to left.

Let us say we want to convert the binary number 10011011_2 to decimal.

- Start at 2^0, evaluating it as "1".
- Increment the exponent by one for each power.
- Stop when the number of elements in the list is equal to the number of digits in the binary number.
- The example number, 10011011, has eight digits, so the list with eight elements would look like this: 128, 64, 32, 16, 8, 4, 2, 1.

EXAMPLE 6.1

Convert 10011011_2 to its decimal format.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	0	0	1	1	0	1	1

The next step is to add the numbers, multiplying them with the base-2 value.

Solution

$(128 \times 1) + (64 \times 0) + (32 \times 0) + (16 \times 1) + (8 \times 1) + (4 \times 0) + (2 \times 1) + (1 \times 1)$
$= 128 + 0 + 0 + 16 + 8 + 0 + 2 + 1$
$= 155$

$10011011_2 = 155_{10}$

EXAMPLE 6.2

Convert 1,12 to its decimal equivalent.

2^0	.	2^{-1}
1	.	0,5
1	.	1

Solution

Now we can evaluate the arithmetic expression.

$1 \times 1 + 0,5$
$1 + 0,5$
$= 1,5^{10}$

$1,1_2 = 1,5_{10}$

6.4.7 Converting a decimal number to a binary number

The steps to convert a decimal number to a binary number are as follows:

- Divide the given number by 2.
- Take the quotient for the next iteration.
- Take the remainder for the binary digit.
- Divide the obtained quotient by 2 again.
- Repeat the steps until you get a quotient equal to 0.

EXAMPLE 6.3

Convert decimal 16 to its binary equivalent.
Note: Copy the value as it is from bottom to top.

2	16	REMAINDER
2	8	0
2	4	0
2	2	0
2	1	0
	0	1

How to convert 112 into the binary number system.
NOTE: Copy the value as it is from bottom to top.

2	112	REMAINDER
2	56	0
2	28	0
2	14	0
2	7	0
2	3	1
2	1	1
2	0	1

$112_{10} = 1110000_2$

You can also use Python to convert the given point number representation to any other point number representation. For example, converting 122_{10} to its binary equivalent would be done as follows:

```
>>> print(bin(112))
0b1110000
>>>
```

Using binary has the advantage of being a base that can easily be represented by electronic devices. Binary numbers are also easier to code, require less computation and have fewer errors in computation. The disadvantage of binary numbers, however, is that it is difficult for us to read and write because of the large number of binary equivalents to one decimal number.

6.4.8 Hexadecimal numbers

Binary digits are used to represent a decimal base-10 number. The equivalent binary string of 1s and 0s can be quite long and confusing. Hexadecimal numbers group binary numbers into sets of four, allowing for the conversion of 16 different binary digits. The hexadecimal number system is a number representation techniques in which the value of the base is 16.

6.4.9 Uses of hexadecimal numbers

The hexadecimal number system is commonly used in computer programming and microprocessors. It is also helpful to describe the colours on web pages. Each of the three primary colours (red, green and blue) is represented by two hexadecimal digits to create 255 possible values.

Using hexadecimal numbers offers the advantage of storing more numbers using less memory. For example, it stores 256 numbers with two digits while decimal numbers store 100 numbers. The major disadvantage of this number systems is that it is difficult to read and write; performing operations such as multiplication and division is also difficult.

6.4.10 Hexadecimal numbers and their decimal equivalents

Using the information provided in the table below, you will see that it is easy to convert binary numbers to hexadecimal, and vice versa.

For example, the binary number 1101 0101 1100 11112 can be converted into an equivalent hexadecimal number of D5CF, which is much easier to read and understand than the long row of 1s and 0s that we had before.

Let us look at the equivalents of hexadecimal, decimal and binary numbers in the following table:

HEXADECIMAL	DECIMAL	BINARY	HEXADECIMAL	DECIMAL	BINARY
0	0	0000	**A**	10	1010
1	1	0001	**B**	11	1011
2	2	0010	**C**	12	1100
3	3	0011	**D**	13	1101
4	4	0100	**E**	14	1110
5	5	0101	**F**	15	1111
6	6	0110			
7	7	0111			
8	8	1000			
9	9	1001			

Table 6.2: Hexadecimal number equivalents

6.4.11 Converting decimal to hexadecimal

The process of converting a decimal number to a hexadecimal one is simple, although it involves more steps than the previous conversion we have looked at.

Step 1: Divide the decimal number by 16.

Step 2: Write the remainder in hexadecimal form.

Step 3: Divide the result by 16.

Step 4: Repeat steps 2 and 3 until the result is 0.

EXAMPLE 6.4

1. Convert 512_{10} to its hex equivalent.

16	512	REMAINDER
16	32	0
16	2	0
	0	2

$512_{10} = 200_{16}$

2. Convert 1128_{10} to its hex equivalent.

16	1128	REMAINDER
16	70	8
16	4	6
	0	4

$1128_{10} = 468_{16}$

You can also use Python to convert the decimal number to hexadecimal as follows:

```
#Covert from decimal to hexadecimal
print(hex(1128))
0×468
```

We have seen that converting hexadecimal to binary is very easy as one simply represents each number using four bits. Here is another example.

EXAMPLE 6.5

1. Convert 7C to its binary equivalent.

HEX	7	C
Binary	0111	1100

$7C_{16} = 0111\ 1100_2$

2. Convert 2B7 to its binary equivalent.

HEX	2	B	7
Binary	0010	1011	0111

$2B7_{16} = 0010\ \ 1011\ \ 0111_2$

Conversion from hexadecimal to decimal number system

There are various indirect and direct methods to convert a hexadecimal number into a decimal number. In an indirect method, you need to convert a hexadecimal number into a binary or octal number, and then into a decimal number. The quickest way is to raise each number to a power of 16, as shown in the example below.

EXAMPLE 6.6

Using the information in Table 6.2, we can deduce the value of each letter from letter A to letter F. Convert hexadecimal number F1 into decimal number.

Solution

Since we know F =15, then: $15 \times 16^1 + 1 \times 16^0$

$240 + 1 = 241$

$F1_{16} = 241_{10}$

Converting from hex to decimal using Python:

```
#Convert from hexadecimal to decimal
print(int('A5', 16))
165
```

Activity 6.4 **INDIVIDUAL ACTIVITY**

1. List THREE examples of positional number systems. (3)
2. Convert the decimal number 57 to a binary number. Show your working. (3)
3. State ONE advantage and ONE disadvantage of hexadecimal numbers. (2)
4. Which of the following is not a type of number system?
   - **A** Positional
   - **B** Non-positional
   - **C** Octal
   - **D** Fractional (1)
5. The *base* is the total number of digits in a number system.
   - **A** True
   - **B** False (1)
6. Convert each of the following hexadecimal numbers into their decimal equivalents.
   - **6.1** 5
   - **6.2** A
   - **6.3** 2B (3 × 4) (12)

TOTAL: 22

6.5 Working with characters and strings

6.5.1 Characters and strings

Python does not have a character or *char type*. All single characters are strings of length one. If you want to use a character, you can just go with str of length 1. You have already seen the operators +, * and the effect they have when applied to operands in section 6.3. The + operator will result in the concatenation of values when applied to strings. The * indicates multiplication in Python. When applied to operands, the * operator creates multiple copies of a string.

```
>>> temp='cold'
>>> temp * 4
'coldcoldcoldcold'
>>>
```

The *in* operator returns *True* if the first operand is contained within the second, and *False* otherwise.

```
>>> myCountry= 'South Africa'
>>> 'Af' in myCountry
True
>>> 'AA' in myCountry
False
>>>
```

Python provides many functions that are built into the interpreter and always available, such as `chr( )`, `ord( )`, `len( )` and `str( )`. You already know how to use `len( )` and `str( )`.

`chr(n)` is used to return a character value for the given integer.

```
>>> chr(102)
'f'
>>> chr(126)
'~'
>>>
```

102 is the ASCII code for the lower-case letter *f*.

`ord(n)` is used to returns an integer value for the given character. `ord( )` does the opposite of `chr( )`. It returns the character represented by the ASCII code given.

ASCII was the first character set (encoding standard) used between computers on the Internet. ASCII is a seven-bit character set containing 128 characters. It contains the numbers from 0 to 9, the upper- and lower-case English letters from A to Z and some special characters.

```
>>> ord('A')
65
>>> ord('#')
35
>>>
```

Letter A has a value 65 on the ASCII table. All characters on the keyboard have an ASCII value.

eLINKS

For more details on ASCII table values, visit the following link:
https://www.asciitable.com/

6.5.2 Initialise data as a character

This section will revise how to create a string of *n* characters. We will create a multiline string and insert a `newline(n)` character. In the example that follows, a new line can easily be displayed by using the `newline(n)` character.

To declare and assign a variable to a string in Python, we can also use single quotes and triple quotes. Normally, single and double quotes are used to assign a string with a single line of characters, but triple quotes are used to assign a string with multiline of characters.

You can put the quotes inside the string by using a backslash (\) directly in front of the quotes. For example:

```
>>>print("Hello, \"I am enjoying Python programming!\"")
```

The output will be as follows:

```
Hello, "I am enjoying Python programming!"
>>>
```

Python allows you to create a multiline string and add a `newline (\n)` character.

```
string = "Java is a programming language\n Sql is a
database"
print(string)
```

The output will be as follows:

```
Java is a programming language
Sql is a database
```

6.5.3 Initialise data as a string

To initialise an empty string in Python, just create a variable without assigning any characters or even using a space. This means assigning " " to a variable to initialise an empty string.

```
empty_string = ""
```

6.5.4 Manipulate character and string data for output purposes

Python escape characters

Using an *escape character* will allow you to insert illegal characters into a string. An escape character is inserted by inserting a backslash (\) followed by the desired character.

Example

```
students='students do like 'Programming' as it prepares
the future jobs'
```

The above code will result in an error. The easiest way to fix the problem is by adding escape characters, as shown below.

```
students = 'students do like \'Programming\' as it
prepares the future jobs'
print(students)
#Output
students do like 'Programming' as it prepares the future
jobs
```

Common escape characters

`\'` – Will result in a single quote

```
students = 'The\'s a group of future coders'
print(students)
The's a group of future coders
```

`\\` – Will insert one backslash

```
students = 'There is \\ a group of future coders'
print(students)
Output
There is \ a group of future coders
students = 'The\'s a group of future coders'
print(students)
The's a group of future coders
```

`\n` – Is used to add newlines to your string

```
>>> print("Multiline strings\ncan be created\nusing escape
sequences.")
Multiline strings
can be created
using escape sequences.
>>>
```

`\t` – Will insert a tab space between the strings

```
students = 'There is \t a group of future coders'
print(students)
Output
There is a group of future coders
```

6.5.5 Built-in functions

Built-in functions are predefined in the programming language's library and can be called directly by the program to achieve specific functional requirements. There are several built-in functions, but we are going to discuss a few.

find( )

The find( ) method returns the lowest index or first occurrence of the substring (if found). If no substring is found, it returns *–1*.

NOTE

The interpreter also counts blank spaces. The staring index is 0.

Example

```
message = 'It is fun learning a programming'
level="NCV Level 2"
#Check the index of 'fun' & 'level
print(message.find('fun'))
print(level.find('Level'))
Output
6
4
```

format( )

The string format( ) method formats the given value(s) in the string into a more effective output in Python.

Syntax:

```
string.format(value1, value2...)
```

The format( ) method takes any number of parameters.

```
>>> myDetails = "My name is {0}, I'm {1}".
format("Siyanda",17)
>>> myDetails
"My name is Siyanda, I'm 17"
>>>
```

Another alternative would be to use empty placeholders { }, as shown below:

```
>>> myDetails = "My name is {}, I'm {}".format("Siyanda",17)
>>> print(myDetails)
My name is Siyanda, I'm 17
```

The third option would be to use variable names:

Syntax:

```
>>> myDetails = "My name is {firstname}, I'm {myAge}".format(
firstname='Siyanda',myAge=17)
>>> print(myDetails)
My name is Siyanda, I'm 17
```

The format function is very important in formatting output when working with numbers such as currency.

Syntax:

Call `str.format(float)` with str as `"R{:,.2f}"` to convert float to a string representation of an amount of money in South African rand. The value 2 is for the number of decimal places.

```
>>> amount=125.68792
>>> print ("The total price is R{:,.2f}".format(amount))
The total price is R125.69
>>>
```

index( )

The `index( )` method finds the first occurrence of the specified value.

```
>>> origin="South Africa –my home country"
>>> print(origin.index('home'))
17
>>> print(origin.index('world'))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: substring not found
>>>
```

As indicated in the code above, if the string does not exist, the interpreter will raise an exception. The word *world* does not exist, hence the *ValueError* exception. In general, `index(< )` and `find( )` are the same method, the only difference being that `find( )` returns –1 if the value cannot be located.

isalpha( )

This function returns *True* if all the characters in the string are letters of the alphabet.

```
>>> course='NCV'
>>> print(course.isalpha( ))
True
>>> course='NCV L2'
>>> print(course.isalpha( ))
False
>>>
```

isascii( )

This function returns *True* if all the characters in the string are ASCII characters.

Example

```
>>> myCharacters='&^%$#@!'
>>> print(myCharacters.isascii( ))
True
```

If we test ISASCII on numbers, an AttributeError exception will be raised, as shown below:

```
>>> myCharacters=5555
>>> print(myCharacters.isascii( ))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'isascii'
>>>
```

isdigit( )

This function returns *True* if all characters in the string are digits. If part of the string is not a number, Python will return *False*.

Example

```
>>> x='78'
>>> print(x.isdigit( ))
True
>>> y='Year 2023'
>>> print(y.isdigit( ))
False
>>>
```

lower( )

This converts a string into lower case.

Example

```
>>> mySubject='ProGRamMing'
>>> print(mySubject.lower( ))
programming
>>>
```

upper( )

This converts a string to uppercase.

Example

```
>>> mySubject='programming'
>>> print(mySubject.upper( ))
PROGRAMMING
>>>
```

Split( )

As the name suggests, this function splits the string at the specified separator and returns a list.

```
>>> mySubject='ProGRamMing is_fun'
>>> print(mySubject.split('_'))
['ProGRamMing is', 'fun']
```

NOTE

The output of the split string is of the list data type.

DID YOU KNOW

You can also implement two methods in one line, e.g. converting the string to uppercase and then splitting it. Here we implement `.upper( )` *and* `.split( )`*:*

```
>>> print(mySubject.upper( ).split('_'))
['PROGRAMMING IS', 'FUN']
```

6.5.6 Concatenate strings

As mentioned in Module 3, *string concatenation* means to add strings together. Use the + character to add a variable to another variable.

To add a space between them, add " ". You can also use single quotes, as illustrated below:

```
>>> sentence='My name is'
>>> firstname='Andrew'
>>> newSentence=sentence + ' ' + firstname
>>> print(newSentence)
My name is Andrew
>>>
```

The + character functions as a mathematical operator for numbers. Python will throw an error if you try to mix a string with a number.

```
>>> sentence='My name is'
>>> mynumber=17
>>> print(sentence + mynumber)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

6.5.7 Converting a string to another data type

In some cases, you may need to convert between built-in data types. You can convert between types by using the type name as a function. There are two types of type conversion in Python:

- Implicit
- Explicit.

During implicit type conversion, the Python interpreter automatically converts one data type to another without requiring the user to intervene. The following examples will help you to better understand the topic:

```
>>> x = 45
>>>
>>> print("x is of type:",type(x))
x is of type: <class 'int'>
>>>
>>> y = 23.6
>>> print("y is of type:",type(y))
y is of type: <class 'float'>
>>>
>>> z = x + y
>>>
>>> print(z)
68.6
>>> print("z is of type:",type(z))
z is of type: <class 'float'>
>>>
```

The variable *z* has automatically been changed to the float type, while the variable *x* has been changed to an integer type, and the variable *y* has been changed to a float type.

With Python's explicit type conversion, the data type is manually changed by the user. We use the predefined functions such as `int( )`, `float( )`, `str( )`, etc. to perform explicit type conversions. This type of conversion is also called *typecasting* because the user 'casts' (changes) the data type of the objects.

```
>>> num_int = 123
>>> num_str = "456"
#Typecasting from string to number (int)
>>> num_str = int(num_str)
>>> print("Data type of num_str after Type
Casting:",type(num_str))
Data type of num_str after Type Casting: <class 'int'>
>>> num_sum = num_int + num_str
>>> print("Sum of num_int and num_str:",num_sum)
Sum of num_int and num_str: 579
>>> print("Data type of the sum:",type(num_sum))
Data type of the sum: <class 'int'>
>>>
```

NOTE

Typecasting may cause data loss if a specific data type is forced on the object.

In the above program, we converted `num_str` from string to integer type using the `int( )` function to perform the addition. After converting `num_str` to an integer value, Python is able to add these two variables to a third variable `num_sum`. We got the `num_sum` value and data type to be an integer.

Activity 6.5 **INDIVIDUAL ACTIVITY**

1. Explain the following built-in functions:

1.1 `find( )`
1.2 `format( )`
1.3 `index( )`
1.4 `isalpha( )`
1.5 `isdigit( )` (5 × 2) (10)

2. Consider the following declaration:

```
>>> myLevel="NCV Level 2 Programming"
```

Write a code snippet to:

2.1 print the length of the string.
2.2 convert all text to capital letters.
2.3 print out the following letters: NCV Level 2 Prog.
2.4 print *True* or *False* after checking whether the string is a digit. (4 × 2) (8)

TOTAL: 18

6.6 Basic file output

Python provides some built-in functions to perform both input and output operations. In this section, we will focus on the output.

6.6.1 Using the open ( ) function

The methods and objects discussed in this section will allow you to save data to external text files. A *text file* is a file where each line of text is terminated with a special character called *EOL* (end of line), which is the newline character (`'\n'`) by default. Text files have a *.txt* extension. Python has several functions for creating, reading, updating and deleting files.

We will start by opening text files using Python. If you want to read a text file in Python, you first have to open it. You can open files with the `Open` function, which has the following syntax:

```
open (name [ , mode [, buffering]])
```

The `Open` function takes a file name as its only required argument and returns a file object. The mode and buffering arguments are both optional and will be explained below.

We are going to discuss six different methods (modes) for opening a file:

MODE	DESCRIPTION	ACTION
'r'	Read only	Opens text file for reading; raises error if file does not exist
'a'	Append only	Opens a file for writing; creates the file if it does not exist; data inserted at the end
'w'	Write	Opens a file for writing; creates the file if it does not exist; data of existing files is truncated and overwritten
'x'	Create	Creates the specified file; returns an error if the file exists
'r+'	Read and write	Opens the file for reading and writing; handle is positioned at the beginning of the file
'w+'	Write and read	Opens file for reading and writing; for existing file, data is truncated and overwritten

We will start by creating a text file called *testing.txt*. The file is empty for now.

```
#Creating a text file in the same location
myNewFile=open('testing.txt', 'x')
```

In addition, you can specify whether the file should be handled as binary or text mode. For binary files, we use 'b' and for text files 't'. The default is text files.

NOTE

There are two types of files that can be handled in Python: normal text files and binary files (written in binary language: 0s and 1s). Text files are used for storing plain text, JSON files and XML. Binary files are used for storing compiled code, app data, media files such as images, etc.

In the previous example, a text file named *testing.txt* was created. Now add the following two lines:

```
Hello students. How is programming
```

The code below will open a text file in PyCharm IDE using Python.

```
#Opening a text file saved in the same project location
myfile = open('testing.txt', 'r+')
print(myfile.read( ))
#Closing the file
myfile.close( )
```

Run the program and the output will be printed. In this case, the text file is called *testing.txt*. The file is opened in *reading and writing mode* as depicted by '*r+*'. This allows you to read from and write in the text file. The *read( )* function provides the entire text by default, but you may specify how many characters to return by adding a value inside the *read( )* bracket, e.g.:

```
print(myfile.read(5))
```

The above code would print: `'Hello'` only and leave out the rest of the text in the file.

If the file is located in a different location, you will have to specify the file path as follows:

```
myfile = open("C:\\myfiles\testing.txt", "r+")
```

Here is another way to open the file:

Syntax:

```
with open('testing.txt') as myfile:
….print(myfile.read( ))
```

The benefit of the code above is that you do not have to implement the close method. Python will close the file even if an exception occurs.

In the last example, we implemented the *read( )* method. Let us explain the different reading methods.

- `read( )` – reads all text from a file into a string; useful if you have a small file and you want to manipulate the whole text of that file
- `readline( )` – reads the text file line by line and returns all the lines as strings
- `readlines( )` – reads all the lines of the text file and returns them as a list of strings.

The output will look as follows:

```
['Hello students. How is programming]
```

6.6.2 Opening a text file for writing

When opening a file for writing, you can use '*a*' (append) mode or '*w*' mode. The write option will overwrite any existing content. This is not advisable if you want to keep existing text. The append mode will open a file and add new text at the end of the file.

Example

```
myfile = open('testing.txt', 'a')
myfile.write("Are you enjoying working with text files")
myfile.close( )
myfile = open('testing.txt', 'r')
print(myfile.read( ))
#Closing the file
myfile.close( )
```

The output will look as follows:

```
Hello students. How is programmingAre you enjoying working
with text files
```

For readability, you could modify the code by adding a '\n' (newline escape character) as shown below:

myfile.write(**"\nAre you enjoying working with text files"**)

```
#Output will then look as follows
Hello students. How is programming
Are you enjoying working with text files
```

6.6.3 Using file object to write to a text file

We previously demonstrated a quicker way to open a file using a file object. The same method will be used to write to a file.

We want to add the following two lines:

Next year I will be in Level 3.

In two years' time I will be doing Level 4.

```
with open('testing.txt', 'a') as myfile:
myob=myfile.write('Next year I will be in Level 3.')
myob=myfile.write('\nIn two years' time I will be doing Level
4.')
myfile2 = open('testing.txt', 'r')
print(myfile2.read( ))
```

Your output will have the two added lines:

```
Hello students. How is programming
Are you enjoying working with text files
Next year I will be in Level 3.
In two years' time I will be doing Level 4.
```

6.6.4 Trapping errors

In this section, we will look at a method for dealing with mistakes. If an exception occurs, the software will not crash; instead an error message will be displayed. In the next example, we will attempt to open a file in reading mode while also attempting to add extra content.

Syntax:

```
try:
  with open("file.txt","r") as myfile:
    #stuff goes here
except IOError:
  #do what you want if there is an error with the file opening
```

Sample program:

```
#opening a text file saved in the same location
try:
  with open('testing.txt', 'r') as myfile:
    myob=myfile.write('I was fun working with text files')
    myob=myfile.write('\nIn two years' time I will be doing
    Level 4.')
    print(myfile.read( ))
except IOError:
  print('Incorrect file mode')
```

The code jumps to display an exception after executing the application because the user opened the file in read mode but attempted to write content.

6.6.5 Closing a file

close( ) method

The file you open will be open until you close it using the *close( )* function. It is critical to close any files that are no longer in use. If the file is not closed, the software may crash or the file may be corrupted. To close the file, use the *close( )* function as shown below:

```
myfile.close( )
```

After closing the file, you can still open it for other operations.

Activity 6.6 **INDIVIDUAL ACTIVITY**

1. Differentiate between a *text file* and a *binary file*. (2)
2. Write the Python statements to open a text file called *example.txt* in both read and write mode. (2)
3. There are THREE basic operations we can perform on a file. Name them. (3)
4. Differentiate between file modes *r+* and *w+* with respect to Python. (4)
5. Whenever possible, what is the recommended way to ensure that a file object is properly closed after use?
   - **A** Making sure that you use the `.close( )` method before the end of the script
   - **B** By using the *try/finally* block
   - **C** It does not matter
   - **D** By using the `with open( )` as statement (1)
6. When reading a file using the file object, what is the best method for reading the entire file into a single string?
   - **A** `.readline( )`
   - **B** `.read( )`
   - **C** `.readlines( )`
   - **D** `.read_file_to_str( )` (1)
7. To open a file *c:|scores.txt* for appending data, we use ...
   - **A** `outfile = open("c:\\scores.txt", "a").`
   - **B** `outfile = open("c:\\scores.txt", "rw").`
   - **C** `outfile = open(file = "c:\scores.txt", "w").`
   - **D** `outfile = open(file = "c:\\scores.txt", "w").` (1)
8. To read five characters from a file object *infile*, we use ...
   - **A** `infile.read(5).`
   - **B** `infile.read.5.`
   - **C** `infile.readline( ).`
   - **D** `infile.readlines( ).` (1)

TOTAL: 15

Summative assessment

1. What will the output of the code below be?

```
>>> x=125; y=13
>>> x//y
```

A 125/13
B 10
C 9
D 9.62 (1)

2. What are the values of the following Python expressions?

```
2**(3**2)
(2**3)**2
2*3*2
```

A 512, 64, 512
B 512, 512, 512
C 64, 512, 64
D 64, 64, 64 (1)

3. What will the output of the following Python code be?

```
>>>list1 = [1, 3]
>>>list2 = list1
>>>list1[0] = 4
>>>print(list2)
```

A [1, 4]
B [1, 3, 4]
C [4, 3]
D [1, 3] (1)

4. Convert 4610 to a binary number. (3)

5. Explain what a *positional* or *weighted system* is. (2)

6. Write a program to create a new string made up of the first, middle and last character of the input string. Given:

```
str1 = "James"
```
(8)

7. Given two strings, *name1* and *name2*, write a program to create a new string *name3* by appending *name2* in the middle of *name1*. (8)

Example:
```
name1 = "Ault"
name2 = "Kelly"
Expected Output
AuKellylt
name1 = "Godwin"
name2 = "Rudo"
Expected Output
GodRudowin
```

8. Write a Python program to display a float number with two decimal places using `print( )`. (2)

Example:
```
num = 458.541315
Output
458.54
```

Summative assessment (continued)

9. Consider the following declarations of two variables *x* and *y*:

```
>>> x=6; y=7
```

Write a program to swap the values of the two variables so that when *x* is printed, the value is 7 and when *y* is printed, the value would be 6. (4)

TOTAL: 30

Module 7

Math, interactive input, constants and errors

After you have completed this module, you should be able to:

- explain what happens when the computer encounters the `input( )` statement;
- use the Python `input( )` function to read user input (numeric) and store the result in a variable;
- use the Python `input( )` function to read user input (single character) and store the result in a variable;
- define the term *validation*;
- explain why user input validation is important;
- convert user input or other values to *Integer* using the `int( )` function;
- convert user input or other values to *Float* using the `float( )` function;
- convert *Integer* and *Float* to *String* using the `str( )` function;
- explain the concept of a *preprogrammed (library) module* in Python;
- explain the need for preprogramed math functions;
- write Python code to import the math module;
- write Python code that makes use of the following common mathematical module functions:
  - `fabs( )`
  - `pow( )`
  - `sqrt( )`
- list the mathematical functions covered and the return type they will produce; and
- write Python code containing complex mathematical expressions and math functions.

Introduction

VOCABULARY

Expression – *string that will be evaluated as Python code*

In this module, you will learn more about Python user input from the keyboard. The input can be read using the `input( )` function, a built-in Python function. This function does not evaluate the **expression**, but simply takes the user input and converts it into a string. You will also be introduced to the math module in Python, a standard module that allows you to use a range of mathematical functions.

7.1 Python keyboard input

In this section, the focus is on the `input( )` function and data validation, which refers to the checks performed by the program to make sure the data meets certain rules or rest.

7.1.1 The *input( )* statement

With the help of examples, you will learn how to use the Python `input( )` function. Developers frequently need to connect with users, either to obtain data or to offer a result. Today, most programs use dialogue boxes to prompt the user for input. Python includes two built-in keyboard-reading routines:

- `input( )`
- raw_input( ).

We are going to cover the input method only, as the raw_input function was only used in the Python 2 version. The syntax of the input function is:

```
input(prompt)
```

The *prompt* string is printed on the console and control is given to the user to enter the value. Some useful information is printed to guide the user to enter the expected value, as shown below:

```
firstName = input("Enter your name:")
print("Your firstname is" + firstName)
```

As soon as the `input( )` function is called, the program execution is stopped until the user provides input. The prompt printed on the output screen to ask the user to submit an input value is optional, i.e. the text or message displayed on the screen is not required, but it provides useful information. After entering the value, the user presses the Enter key. The input function returns a string (as shown above) that can be stored as a variable, as you will learn later.

7.1.2 Using the Python *input( )* function to read numeric user input

By using the input function, whatever you submit as input will be converted into a string. Even if you submit an integer value, the `input( )` method will convert it to a string. There is no way to get an integer or any other type as the user input. However, we can use the built-in functions to convert the entered

string to an integer. In your code, you must explicitly convert it to an integer using the typecasting method, e.g.:

```
firstNumber = int(input("Enter the firstNumber:"))
print("Your first number is" + str(firstNumber))

secondNumber = int(input("Enter the secondNumber:"))
print("Your second number is" + str(secondNumber))
answer = firstNumber + secondNumber
print(firstNumber, '+', secondNumber, '=', answer)
```

The last line of the above statement can be used with the + as the concatenating operator as follows:

```
>>>print(str(firstNumber) + '+' + str(secondNumber) + '=' +
str(answer))
```

We have already mentioned what will happen if you do not typecast from string to integer: concatenation of the values. The alternative would be to use the `eval( )` function. This function evaluates the string like a Python expression and returns the result as an interger. The syntax of the eval function is:

```
eval(expression, [globals[, locals]])
```

For example:

```
>>>eval("4+9")

Output
13
>>> eval("sum([3,4,5])")
12
```

VOCABULARY

Global – *dictionary containing global parameters with scope throughout the program*
Local – *dictionary containing local parameters as a mapped object*

A Python expression can be evaluated based on either a string input or a compiled code input using `eval( )`. You can use the `eval( )` function to evaluate Python expressions dynamically from string or compiled code inputs.

The code in the previous example can be modified as follows using the *eval* function:

```
firstNumber = eval(input("Enter the firstNumber:"))
print("Your first number is" + str(firstNumber))

secondNumber = eval(input("Enter the secondNumber:"))
print("Your second number is" + str(secondNumber))
answer = firstNumber + secondNumber
print(str(firstNumber) + '+' + str(secondNumber) + '=' +
str(answer))
```

DID YOU KNOW

`Eval( )` *works on strings, Boolean, logical (Boolean), membership, identity and comparison operators. Strings must be one block, as shown below:*

```
>>> eval("5"+"8")
58
>>> eval("5+8")
13
>>> number1=25; number2=34
>>> eval("number1 != number2")
True
>>> eval("7 >6")
True
>>> eval("5>=3") #comparison operator
True
>>> x=[5,6,3]
>>> eval("3 in x") #membership operator
True
>>>
>>> eval(5+8) # will give an error
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: eval( ) arg 1 must be a string, bytes or code
object
```

You can use `eval( )` with Boolean expressions that use any of the following Python operators:

- Value comparison operators: <, >, <=, >=, ==, !=
- Logical (Boolean) operators: *and*, *or*, *not*
- Membership test operators: *in*, *not in*
- Identity operators: *is*, *is not*.

Assignment operations are not allowed with `eval( )` either. Refer to the example below:

```
>>> eval("number1 =23")
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "<string>", line 1
  number1 =23
      ^
SyntaxError: invalid syntax
>>>
```

You can pass any kind of math expression into `eval( )` to evaluate it. If everything is correct, `eval( )` will give you the expected result after parsing (analysing it into logical syntactic components) and evaluating it.

```
>>> eval("(12 + 6) / 2")
9.0
>>>
```

Let us look at the *datetime* module. Remember, a module is a file containing Python statements and definitions. Think of it as a code library containing a set of functions you want to include in your application. To work with dates as date objects in Python, we can import the *datetime* module.

For example:

```
import datetime
#now method returns the current date and time.
todays_date = datetime.datetime.now( )
print(todays_date)

#Output
2022-07-07 21:09:02.859215
```

Note that the output in the above example contains the year, month, day, hour, minute, second and microsecond. The *datetime* module has many methods to return information about the date object. In the example below, we use the *datetime* module to return the current year, month and name of the weekday.

```
import datetime
#now method returns the current date and time.
todays_date = datetime.datetime.now( )
#using the year method to extract the current year
print(f'The current year is {todays_date.year}')
#using the year method to extract the current day
print(f'The date today is {todays_date.day}')
#using the year method to extract the current month
print(f'The current months is {todays_date.month}')

#Output
The current year is 2022
The date today is 7
The current months is 7
```

The above code can be rewritten as follows:

```
from datetime import date
x=date.today( )
#using the year method to return the current year
print(f'The current year is {x.year}')
#using the year method to return the current day
print(f'The date today is {x.day}')
#using the year method to return the current month
print(f'The current months is {x.month}')
```

Note the first line where *import* is implemented. In the example above, we start with:

```
from <module> import …
```

The distinction between *import* and *from* <*module*> *import* in Python is:

- an *import* imports the whole library; while
- a *from* <*module*> *import* imports a specific member or members of the library.

In the examples we have looked at, note that the month and day are numeric values instead of names such as 'July' and 'Thursday'. You can format the output to show strings by using the method called the `strftime( )`, which takes one parameter within the brackets, as illustrated below:

Syntax:

```
#formatting dates
import datetime
x = datetime.datetime.now( )
print(x.strftime("%B"))
```

Compare the syntax used here to the example below:

```
#formatting dates
import datetime
current_date = datetime.datetime.now( )
print(current_date.strftime("%Y")) #return full year
e.g. 2022
print(current_date.strftime("%y")) #return last 2 digit of
the year e.g. 22
print(current_date.strftime("%A")) #return the day in full
e.g. Thursday
print(current_date.strftime("%a")) #return the day in short
e.g. Thur
print(current_date.strftime("%B")) #return the month in full
e.g. July
print(current_date.strftime("%b")) #return the month in
short e.g. Jul

#Output
2022
22
Thursday
Thu
July
Jul
>>>
```

Let us analyse the common formatting codes used:

- `%A` – gives the full day, e.g. Thursday
- `%a` – returns the day in short, e.g. Thu
- `%B` – gives the full month, e.g. July
- `%b` – returns the month in short, e.g. Jul
- `%Y` – gives the year in full, e.g. 2022
- `%y` – returns the short version of the year, e.g. 22.

eLINKS

To learn more about formatting reference, visit the following link:
futman.pub/Python_dates

EXAMPLE 7.1

Here is a challenge for you. Check what day of the week it will be five days from now (or any given day).

Solution

```
from datetime import datetime
#now method returns the current date and time.
todays_date = datetime.now( )
#using the year method to return the current year
print(f'The current year is {todays_date.year}')
#using the year method to return the current day
print(f'The date today is {todays_date.day+5}')
#using the year method to return the current month
print(f'The current months is {todays_date.month}')
newDate=datetime(todays_date.year,todays_date.month,
todays_date.day+5)
print(f'The date after 5 days will be {newDate}')
```

```
The output will be:
The current year is 2022
The date today is 12
The current months is 7
The date after 5 days will be 2022-07-12 00:00:00
```

Activity 7.1 **INDIVIDUAL ACTIVITY**

Create a program that asks the user to enter his/her name and current age. Print out a message, addressed to the user, that tells him/her the year in which he/she will be 100 years old.
Note: Use the *datetime* module to determine the current year.

The output should be as follows:

```
The current year is 2022-07-07
What is your name: Godwin
How old are you: 45
Godwin, you will be 100 years old in the year 2077
>>>
```

TOTAL: ~~XX~~

[there are no marks?]

7.1.3 Using the Python *input( )* function to read single-character user input

You will now learn how to take only a single character as an input in Python. The easy and fun examples that follow should help you understand the input method. The figure on the next page shows the first step in creating a variable that holds the user's input.

```
1 characterChoice=input('Enter the choice of character: \n')[0]
2 print(characterChoice)
3 print(len(characterChoice))
4
Running: onecharacter.py
Enter the choice of character:
Programming
P
1
>>>
```

Figure 7.1: Character input

As you can see, after the prompt statement, we appended an index to be extracted from the string. The first position of the string is '0'. Our input was 'Programming', but the character stored is only the letter *P*.

EXAMPLE 7.2

Write a Python code to store the '!' (exclamation mark) in a variable if only the first character of the string entered is the '!'.

```
#Checking for vowels
vowels='!$#*^'
character = input('Enter a character\n')[0]
#checking if first character is a vowel
print(character in vowels)

#Output is of type boolean (True) if the input matches
what is expected otherwise the answer is False
Enter a character
!
True
>>>
```

7.1.4 Define the term *validation*

When we accept user input, we must ensure that it is correct. **Validation** is an automatic computer check to ensure that the data entered makes sense and is reasonable. A program checks the data to make sure it meets certain rules or restrictions. In other words, validation ensures that the data is what we expect it to be.

VOCABULARY

Validation – *checking or verifying data against a given set of rules before it enters the computer system*

7.1.5 Importance of input validation

Validating data is crucial for accuracy, quality and integrity. It also ensures that the data collected from different sources meets business requirements.

Some benefits of data validation include the following:

- It ensures cost-effectiveness, because it saves time and money by ensuring that the datasets collected and used in processing are clean and accurate.
- In addition to being easy to integrate, it is compatible with a wide variety of processes.
- It ensures that the data collected from different sources – structured or unstructured – meets business requirements by creating a standard database and cleaning dataset information.
- Increased data accuracy leads to increased profitability and reduced loss in the long run.
- It improves decision making, strategy and market objectives.

There are many different data validation checks that can be done. We may check that the data:

- is of the correct data type, e.g. a number and not a string;
- does not contain invalid values, such as including a letter in an area code;
- is not out of range, e.g. age given as a negative number or there is division by zero;
- meets constraints, e.g. a given date can only be in the future or the message does not exceed the maximum length;
- is consistent with other data or constraints, such as students' test scores and letter grades;
- is valid, e.g. a given file name is used for an existing file;
- is complete, i.e. ensuring that all required form fields have data.

We must ensure that the data type and range are valid. Check all input for each method when the code is contained in a library where anyone can use it. When entering data directly, ensure that the user's key and value are correctly typed and that the range is correct.

Types of validation in Python

There are three types of validation in Python:

- **Type check:** to check the given input data type, e.g. integer, float, etc.
- **Length check:** to check the length of a given input string
- **Range check:** to check whether a given number falls in-between the two numbers used, e.g. the age of a person can be within the range of 0 to 130 years, while his/her height will range from 0,2 cm to under 3,0 metres.

The user can implement in-built validation tools such as Cerberus, Colander, Voluptuous, Valideer and many others. Users can also make use of flags, `try ... except` statements, `isdigit( )`, `isascii( )`, loops, etc. If validation fails, we do not want the program to crash. We therefore need to handle the errors when they occur. We can do this by using `try except` and `if` statements.

try … except syntax

```
try:
 [code to execute if no error is detected]
except:
  [code to execute if an error is detected]
```

- The `try` block lets you test a block of code for errors.
- The `except` block lets you handle the error.
- The `else` block lets you execute code when there is no error.

Consider the following code for adding two numbers: The numbers are checked at entry and if they fail the test, the code jumps to print the statement in the `except` block.

```
try:
  #typecasting of input to integer
  firstNumber=int(input("Enter the firstNumber"))
  secondNumber=int(input("Enter the secondNumber"))
except ValueError:
  #only executes if wrong input is typed in
  print("Something went wrong, program is terminating")
else:
  #executed only when typecasting passes
  print("All the input passed the checks")
  answer=firstNumber+secondNumber
  print(str(firstNumber) +'+' + str(secondNumber) + '=' +
  str(answer))
```

After entering values, the output is as follows:

```
Enter the firstNumber5
Enter the secondNumber7
All the input passed the checks
5+7=12
>>>
```

Using a flag variable

A flag variable can be used to let one part of your program know when something happens in another part of the program. The flag variable is of the Boolean data type. In the example below, a flag variable and a loop are used. (Loops will be discussed in detail in Module 9.)

```
isAdult= False
while not isAdult:
  try:
    firstname=input("Enter your firstname\t")
    surname=input("Enter your surname\t")
    age = int(input('How old are you? \t'))
  except:
    print('You must enter a valid number between 13 and 19')
  else:
    if age >= 18:
      isAdult = True
```

```
    else:
      isAdult=False
      print('You do not qualify to get a drivers licence')
print('Congratulations \t' + firstname +' ' + surname +' ' +
'You now qualify to get a drivers licence')
```

A `try ... except` block can be nested within a while loop, as shown in the code above.

Check whether input contains digits using a flag variable

```
agecheck = False
while not agecheck:
    age = input('How old are you?')
    if age.isdigit( ):
      agecheck = True
    else:
      print('You must enter a valid number')
print('You are' + str(age))
```

Validation of the length of a string

To ensure that strings are of an acceptable length, it is sometimes necessary to validate the length of strings. We can implement the `len( )` method.

```
#Validating length of a password
isLongEnough = False
while not isLongEnough:
  password = input('Enter password of at least 5 characters
for the string:')
  if len(password) >= 5:
    isLongEnough = True
  else:
    print('Password entered is too short')
print('Your password entered is:' + password)
```

7.1.6 Converting user input or other values to integers

To convert (or *cast*) a string to an integer in Python, you use the built-in `int( )` function. The function takes in the initial string you want to convert as a parameter and returns the integer equivalent of the value you passed. The general syntax looks something like this: in ("*str*").

Let us take the following example, where there is the string version of a number:

```
#string version of the number 7
x="7"
print(type(x)) #will print str
#casting to integer
new_value=int(x)
print(type(new_value)) #will return integer
y=7.0
print(type(y)) #returns float
print(type(int(y))) #casting float to int results in integer
```

`int( ):` This function takes a float or string as an argument and returns an integer-type object.

7.1.7 Converting user input or other values to float

It should be noted that the values that can be casted to integer or float must be in the numeric category.

`float( ):` This function takes an interger or string as an argument and returns a float-type object.

Consider the following example:

```
num_int = 123
num_flo = 1.23
newnumber = num_int + num_flo #addition of in to a float
results in float data type
print("datatype of num_int:",type(num_int)) #will return int
data type
print("datatype of num_flo:",type(num_flo)) #will return float
data type
print("Value of new number:", newnumber)
print("datatype of num_new:", type(newnumber))
```

When we run the above program, the output will be:

```
datatype of num_int: <class 'int'>
datatype of num_flo: <class 'float'>
Value of new number: 124.23
datatype of new number: <class 'float'>
>>>
```

In this program, the data type of `num_int` is an integer, while the data type of `num_flo` is a float. We can see that `num_new` has a float data type because Python always converts smaller data types to larger data types to avoid the loss of data.

7.1.8 Converting float to string

Converting a value from float to string requires the use of the `str( )` method.

```
pi = 3.1415
print(type(pi)) #float
piInString = str(pi) #float -> str
print(type(piInString)) #str

Output
<class 'float'>
<class 'str'>
```

Python provides built-in methods for converting data types. You will be able to write your programs with more flexibility if you can work with different data types.

Activity 7.2 **INDIVIDUAL ACTIVITY**

1. Which method is used for inputting data from the user through the keyboard? (1)
2. What will the output of the following code be?

```
print(eval("8"+"9"))
```

Give a reason for your answer. (2)

3. What will the output of the following Python snippet be?

```
number1=34; number2 =54
print(eval("number1 != number2"))
```

(2)

4. True or false: Assignment operators can be used on `eval( )` function. (1)
5. Define the term *module* as applied in Python programming. (2)
6. What will the output of the following Python expression be?

```
round(4.576)
```

A 4.5
B 5
C 4
D 4.6 (1)

7. What will the output of the following Python expression be?

```
round(6.5)
```

A 6.6
B 6.5
C 7
D 6 (1)

8. The function `pow(x,y,z)` is evaluated as:

A `(x**y)**z`
B `(x**y) / z`
C `(x**y) % z`
D `(x**y)*z` (1)

9. What is the output of the following code?

```
import datetime
x = datetime.datetime.now( )
print(x.strftime("%y"))
```

(2)

10. What will the output of the following Python function be?

```
print(min(max(False,-3,-4), 2,7))
```

(1)

TOTAL: 14

7.2 Python mathematical library functions

Python has a built-in module that can be used for mathematical tasks. The *math module* has a set of methods and constants.

7.2.1 Preprogramed (library) module in Python

In this section, you are going to learn more about modules in Python language. In Python, *modules* are simply files with a .py extension containing Python code that can be imported into another Python program. Modules

allow us to logically organise our Python code, which makes the code easier to understand and use.

Like many other programming languages, Python supports *modularity*. That means you can break a large code into smaller and more manageable pieces. And through modularity, Python supports *code reuse*. You can import modules in Python into your programs and reuse the code therein as many times as you want. In the Python language, a module is just a file consisting of Python code that has several built-in or user-defined functions.

Built-in modules

The Python standard library is one of its many superpowers. Many modules are built into this rich standard library. Because of this, the code can be reused often. Python contains modules such as “os”, “sys”, “datetime”, “random”, “math” and “requests”. You can also check the available built-in modules by typing *help* , as shown below:

```
>>>help('modules')
```

You can import and use any of the built-in modules using the *import keyword*.

User-defined modules

Another superpower of Python is that it allows you to take control. Functions that we define ourselves to perform certain tasks are referred to as *user-defined functions*. Let us create a module. Type the following and save it as *mymodule.py*.

```
#Python Module to add numbers
def add(a, b):
  """This program adds two
  numbers and return the result"""
  result = a + b
  return result
```

Here we have defined a function `add( )` inside a module named *example*. The function takes in two numbers and returns their sum. We use the import keyword to do this. To import our previously defined module *mymodule*, we type the following into the Python prompt:

```
>>>import mymodule
```

By using the module name, we can access the function using the dot operator (`.`). For example:

```
import mymodule
result=mymodule.add(6,7)
print(result)

#Output
13
>>>
```

When we import a Python module, the Python interpreter searches for it in the following sequences. If that module is not located in the current directory,

Python searches each directory in the PYTHONPATH variable. If that fails, Python checks the default path. The search path for a module is stored in the system (*sys*) module as the *sys.path* variable. The *sys.path* variable contains the current directory, PYTHONPATH, and the installation-dependent default.

7.2.2 Importance of preprogramed math functions

Modules are important to programmers because they do not have to work on already provided codes. It speeds up the development of the coding process. Some of the advantages of working with modules in Python include the following:

- Reusability – using modules makes the code reusable
- Simplicity – modules focus on a small portion of the problem, rather than on the entire one
- Scoping – a separate namespace can be defined by a module, which helps to avoid collisions between identifiers.

7.2.3 Using the *import math* module

As mentioned earlier, the Python language comes with a mathematical module that can be used to perform mathematical operations. The math module contains a set of methods and constants.

Some of the common math functions include the following:

- `math.pow( )`
- `math.trunc( )`
- `math.prod( )`
- `math.floor( )`
- `math.factorial( )`
- `math.floor( )`
- `math.trunc( )` – returns the truncated integer parts of a number
- `math.floor( )` – rounds a number down to the nearest integer
- `math.factorial( )` – returns the factorial of a number.

Let us implement the math module to calculate the factorial of a number. *Factorial* is a non-negative integer that is the product of all positive integers less than or equal to the number you ask for. It is denoted by an exclamation mark (`!`).

```
n! = n* (n-1) * (n-2) *........1
4! = 4x3x2x1 = 24
```

EXAMPLE 7.3

```
import math
#accepting input
x=int(input("Please enter the integer number\t"))
#calculating the factorial using the math.factorial
result=math.factorial(x)
#displaying output
print(str(x) + "\tfactorial is\t" + str(result))
```

EXAMPLE 7.3 (CONTINUED)

```
#Output
Please enter the integer number 6
6 factorial is 720
>>>
```

This example illustrates a faster method for calculating factorials instead of typing the following code:

Result = 6 × 5 × 4 × 3 × 2 × 1

The factorial function is called from the math module.

We can import modules in different ways to make our code more Pythonic. Using the `import...as` statement allows you to assign a shorter name to a module while using it in your program. We can write the above code as follows:

```
import math as r
#accepting input
x=int(input("Please enter the integer number\t"))
#calculating the factorial using the math.factorial
result=r.factorial(x)
#displaying output
print(str(x) + "\tfactorial is\t" + str(result))
```

Look at the import statement used in the code above. The string *r* is used as a shorter name of math.
We can also use the `from <module> import` statement.

You can import a specific function, class or attribute from a module rather than importing the entire module. Follow the syntax below:

```
from <modulename> import <function>
```

Consider the following example:

```
>>> from random import randint
>>> randint(20, 100)
69
```

If you need to import everything from a module and you do not want to use the dot operator, you can use `import *`. The asterisk denotes *all*.

For example:

```
from math import *
print(factorial(3))
print(sqrt(100))

#Output
6
10.0
>>>
```

Constants

A *constant* is a type of variable whose value cannot be changed. It is helpful to think of constants as containers that hold information that cannot be changed later. Think of a bag that stores some books that cannot be replaced once they have been placed inside the bag.

Assigning value to constants in Python

In Python, constants are usually declared and assigned in a module. Here, the module is a new file containing variables, functions, etc. which is imported to the main file. Inside the module, constants are written in capital letters and underscores are used to separate the words, as it helps us to write more semantic rather than static code.

EXAMPLE 7.4

Declaring and assigning value to a constant

Create a *constant.py* with the following values:
PI=3.14
GRAVITY=9.8

Now you can import the constant file as illustrated below:

```
import constant
print(constant.PI)
print(constant.GRAVITY)

#Output
3.14
9.8
```

eLINKS

For more information on math in Python, visit the following link:
futman.pub/Math_in_Python

7.2.4 Code using common mathematical module functions

We have already discussed how to implement the *import math* module. The same concept will be used to implement the `fabs( )`, `pow( )` and `sqrt( )` functions.

`fabs( )` – the `math.fabs(x)` function returns the absolute value of *x* as a float. *Absolute* denotes a non-negative number. This removes the negative sign of the value if it has any.

For example:

```
>>> import math
>>> math.fabs(66.46)
66.46
>>> math.fabs(7)
7.0
>>> math.fabs(-5)
5.0
>>>
```

`pow( )` – The `math.pow( )` function returns the value of *x* to the power of *y*.

For example:

```
>>> import math
>>> math.pow(5,2)
25.0
>>> math.pow(4,3)
64.0
>>>
```

`sqrt( )` – The `math.sqrt(x)` function returns the square root of *x*.

For example:

```
>>> import math
>>> math.sqrt(25)
5.0
>>> math.sqrt(5.25)
2.29128784747792
```

7.2.5 Mathematical functions

In this section, we are going to look at more mathematical functions within the math module.

`math.ceil(x)` – This function returns the ceiling of *x*, the smallest integer greater than or equal to *x*. In other words, it rounds the number up to the nearest integer, if necessary, and returns the result. If *x* is not a float, delegates to *x.__ceil__*, which should return an integral value.

```
>>> import math
>>> math.ceil(5.2)
6
>>> math.ceil(5.7)
6
```

`math.floor( )` – This function returns the floor of *x*, the largest integer less than or equal to *x*. In other words, it rounds the number down to the nearest integer, if necessary, and returns the result. If *x* is not a float, delegates to *x.__floor__*, which should return an integral value.

```
>>> math.floor(5.7)
5
>>> math.floor(5.2)
5
>>>
```

VOCABULARY

Iterable – *object that can be looped over or iterated over with the help of a for loop*

`math.fsum(iterable)` – This function returns an accurate floating-point sum of values in the **iterable**. It avoids loss of precision by tracking multiple intermediate partial sums, as shown in the example below:

```
>>> mynumbers=[3,5,6,7,8,12,67,23,9.3, 2.4]
>>> sum(mynumbers)
142.70000000000002
>>> math.fsum(mynumbers)
142.7
>>>
```

`math.isnan(x)` – This function returns `True` if *x* is a *nan* (not a number) and `False` otherwise.

```
>>> math.isnan(7)
False
>>> print (math.isnan (float("nan")))
True
>>>
```

`math.prod(iterable, *, start=1)` – This function is used to calculate the product of all the elements in the input iterable. The default start value for the product is 1.

```
>>> x=[4,3,2]
>>> math.prod(x)
24
```

`math.trunc( )` – This function returns the truncated integer parts of a number. `trunc( )` does the same as *math.floor*, as shown in the snippet below:

```
>>> math.trunc(5.2)
5
>>> math.trunc(5.7)
5
>>> math.floor(5.2)
5
>>> math.floor(5.7)
5
>>>
```

Python also uses math constant functions such as *math.pi*.

```
import math
x=math.pi
print(x)

#Output
3.141592653589793
>>>
```

The table below shows other useful built-in functions used in Python.

FUNCTION	DESCRIPTION	CODE
`chr( )`	Returns a character that represents the specified unicode	`x = chr(98)` represents the unicode B

`sorted( )`	Returns a sorted list of the specified iterable object	`letters = ("z", "k", "y", "a", "f", "m", "s", "b")`
		`sorted_list = sorted(letters)`
		Output
		`['a', 'b', 'f', 'k', 'm', 's', 'y', 'z']`

`round( )`	Returns a floating-point number that is a rounded version of the specified number, with the specified number of decimals	`print(round(3.4))`
		Output
		`3`

Table 7.1: Other built-in functions

7.2.6 Complex mathematical expressions and math functions

In this section, you are going to use math modules to evaluate complex mathematical expression.

EXAMPLE 7.5

Write a Python program to calculate the surface volume and area of a sphere.

The area of a sphere is `A = 4*π*r2`

A sphere is a perfectly round geometrical object in 3D space that is the surface of a completely round ball. In three dimensions, the volume inside a sphere is derived to be `V = 4/3*π*r3`, where r is the radius of the sphere.

Solution

```
#Calculating volume of a sphere
import math
#enter value of radius
#A = 4*π*r2
#V= 4/3*π*r3
radius=float(input("Enter value of radius\t"))
#calculating surface area
area=4 *math.pi*math.pow(radius, 2)
#calculating the volume of a sphere
volume=4/3* math.pi * math.pow(radius,3)
print(f'The volume of a sphere is {volume}', f'whose
radius is {radius}')
print(f'The area of a sphere is {area}', f'whose radius
is {radius}')
```

Solution (continued)

```
#Output sample
Enter value of radius .75
The volume of a sphere is 1.7671458676442584 whose radius
is 0.75
The area of a sphere is 7.0685834705770345 whose radius
is 0.75
>>>
```

The output can be further formatted to two decimal places by implementing the `format( )` method as follows:

```
print(f'The volume of a sphere is {"{:.2f}".
format(volume)}', f'whose radius is {radius}')
print(f'The area of a sphere is {"{:.2f}".format(area)}',
f'whose radius is {radius}')

The output will be as follows:
Enter value of radius .75
The volume of a sphere is 1.77 whose radius is 0.75
The area of a sphere is 7.07 whose radius is 0.75
>>>
```

EXAMPLE 7.6

Write a Python program to convert *degrees* to *radians*.

NOTE

The radian is the standard unit of angular measure used in many areas of mathematics. An angle's measurement in radians is numerically equal to the length of the corresponding arc of a unit circle; one radian is just under 57,3 degrees (when the arc length is equal to the radius).

Radian = [[pi/180°]]x degrees

Solution

```
#converting radian
import math
degree = float(input("Input degrees:"))
radian = degree*(math.pi/180)
print(radian)
```

EXAMPLE 7.7

Calculate the area of a triangle whose three sides are of different lengths.

$1/4\sqrt{(b+c+a)(b+c-a)(a+b-c)(a-b+c)}$

In geometry, Heron's formula (named after Hero of Alexandria) gives the area of a triangle when the lengths of all three sides are known. Unlike other triangle area formulae, there is no need to calculate angles or other distances in the triangle first.

We know that the semi-perimeter of a triangle is: $s = (a + b + c)/2$.
From this: $a + b + c = 2s$

So the resultant formula can be re-written as:

$=\sqrt{s(s-a)(s-b)(s-c)}$

Solution

```
import math
#Enter the three sides of the triangle
side_a=float(input("Enter length of first side"))
side_b=float(input("Enter length of second side"))
side_c=float(input("Enter length of third side"))
#Calculate the length of semi-perimeter
side=float((side_a+side_b+side_c)/2)
#calculate the area of the triangle
area=(side*(side-side_a)*(side-side_b)*(side-side_c))
result=math.sqrt(area)
#display the output
print(f'The area of a triangle whose sides are {side_a},
{side_b}, {side_c} is {"{:.2f}".format(result)} cm^2')
```

#Sample output

```
Enter length of first side8
Enter length of second side11
Enter length of third side13
The area of a triangle whose sides are 8.0, 11.0, 13.0 is
43.82 cm^2
>>>
```

There is another library that is very useful for machine learning. NumPy is the fundamental package for scientific computing in Python. NumPy is a Python library used for working with arrays. It also has functions for working in the domains of linear algebra and matrices. To use this library, you will need to import NumPy.

EXAMPLE 7.8

Consider the following marks recorded for 10 students who sat for a Python test:

```
Test_1= [56,78,97,34,76,65,45,77,72,81]
```

Calculate the mean, mode and median for the test marks.

NOTE

The mean value is the average value. To calculate the mean, find the sum of all values, and divide the sum by the number of values. The median value is the value in the middle, after you have sorted all the values. The mode is the value that appears most in the list. The Numpy and SciPy modules have a method for these.

Solution

Mean and median:

```
import numpy
test_1= [56,78,97,34,76,65,45,77,72,81]
#Calculating the mean score
averageScore= numpy.mean(test_1)
#Calculating the median score
middleScore = numpy.median(test_1)
#displaying the output
print("The mean score is \t", averageScore)
print("The median score is \t", middleScore)
```

Mode:

```
from scipy import stats
test_1= [56,78,97,76,76,65,45,77,72,81]
high_occur = stats.mode(test_1)
print("The median score is \t", high_occur)
```

If there is only one occurrence of each score, the result will be the least score. In the above example, the answer is 76. It is important to understand the concepts *mean*, *median* and *mode*, which are commonly used in machine learning.

Python's math module is very useful in dealing with arithmetic expressions, but it is not equipped to handle complex numbers. The Python math module is complemented by the *cmath* module, which implements many of the same functions, but for complex numbers. However, *cmath* does not fall within the scope of this module.

Activity 7.3 **INDIVIDUAL ACTIVITY**

1. What is output of `print(math.pow(3, 2))`?
   - **A** 9.0
   - **B** None
   - **C** 9
   - **D** None of the above (1)
2. Using an example, explain what the function `math.prod( )` is. (3)
3. Write a Python program to calculate the area of a parallelogram.

   Note: A parallelogram is a quadrilateral with opposite sides parallel (and therefore opposite angles equal). A quadrilateral with equal sides is called a *rhombus*, and a parallelogram whose angles are all right angles is called a *rectangle*.

   Hint: Area of parallelogram = base × height

   Show output correct to TWO decimal places. (10)

TOTAL: 14

Summative assessment

1. Explain the effect of the `print( )` statement in Python. (2)
2. Explain how you can ensure that the input is converted to integer when captured. (2)
3. What method would you use to check the length of a string? (2)
4. Define the term *validation* as used in Python programming. (2)
5. List THREE reasons why input validation is important in programming. (3)
6. Identify the THREE types of validation in Python. (3)
7. What is the function of `floor( )`?
   - **A** It returns the next integer of the floating number.
   - **B** It returns the previous integer of the floating number.
   - **C** It returns the floating-point number +1.
   - **D** There is no such function. (1)
8. Write a Python program to check whether the user is a teenager based on the age supplied. The program must use a flag and exception to validate the type of input given by the user and determine whether it lies within a given range. #Range check. (10)
9. List THREE ways to implement *import* in Python. (3)
10. Define the term *exception* as used in Python. (2)

TOTAL: 30

Module 8

Selection control structure

After you have completed this module, you should be able to:

- use the different programming constructs and operators mastered with Scratch or other Python IDEs;
- list and explain the different Python relational operators;
- write Python code that will save the result of relation expressions that contain both or a mix of numeric variables and literals;
- explain the purpose of the different Python logic operators AND, OR and NOT;
- write Python code that will save the result of relation expressions that contain two or a mix of numeric variables, character variables and literals;
- identify or correct the general form for a Python *if-else* statement;
- write Python code that will use relational expressions and logic operators in *if-else* statements;
- identify or correct the general form for a nested *if-else* statement;
- write Python code that will use relational expressions in nested *if-else* statements;
- identify or correct the general form for a Python *if-else* chain; and
- write Python code that will use an *if-else* chain.

Introduction

Relation and logic form the foundation of a program and define its functionality. These fundamentals determine the flow of execution as well as the conditions that should be kept to maintain the flow. In this module you will learn more about the relational expressions, selection statements and logical operators available in Python language. However, before moving on to these new topics, you need to revise important problem solving concepts used in Scratch (Module 3) and the basic Python programming language (Module 5).

8.1 Reviewing generic concepts and problem solving techniques

In Module 3, you learned how to use different types of operators to evaluate expressions. We are going to revise problem solving techniques by using Scratch to solve the following problem:

> Use Scratch to write a program to determine whether a person is eligible to vote or not. If the age entered by the user is greater than or equal to 18, the program should inform the user that he/she is eligible to vote by turning on a green LED. Alternatively, the red LED will turn on if the person is below the legal voting age.

By now you are familiar with GPIO pins (discussed in Module 3). We are going to use GPIO (22) for the green LED and GPIO (27) for the red LED. The completed Scratch code is shown below.

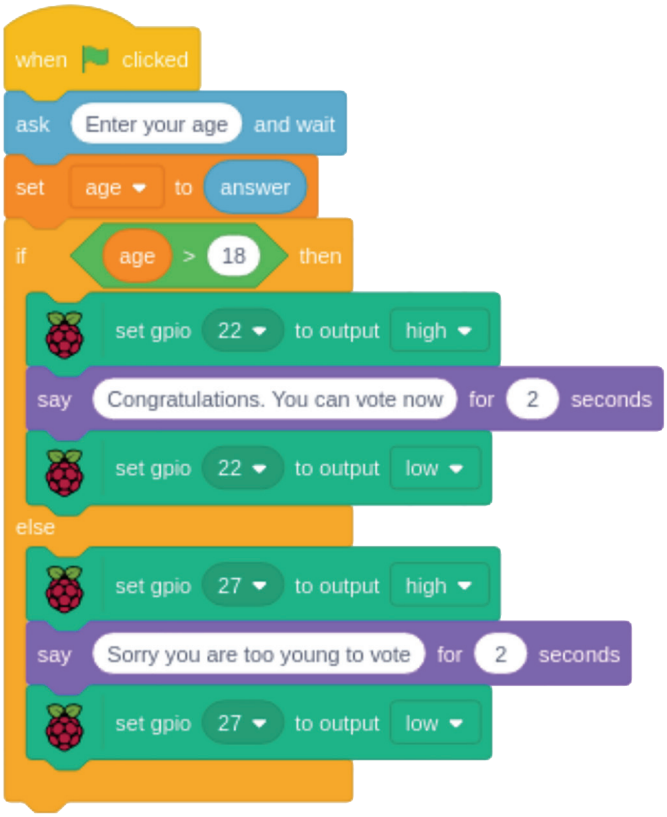


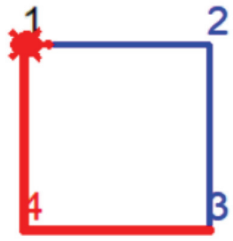
Figure 8.1: Selection structures

Activity 8.1 **INDIVIDUAL ACTIVITY**

Write a program to draw a square using the Python Turtle module. The pen should set the colour of the first two lines in blue and the last two in red.

The output should be as follows:

1. From 1 to 3 the line colour must be blue.
2. From 3 to 1 the line colour must be red. (20)



Square with two line colours

TOTAL: 20

8.2 Conditional tests and logic operators

Conditional statements allow for the conditional execution of a statement or group of statements based on the value of an expression. They allow Python to evaluate the code to see whether it meets the specified conditions. Examples include the *if* and *if-else* statements that will be used in this section. Logical operators are used on conditional statements and perform logical *and*, *or* and *not* operations.

8.2.1 Python relational operators

VOCABULARY

Relational operators – *symbols used to compare the operands on either side and determine the relationship between them*

Relational operators (also called *comparison operators*) are symbols that perform operations on data and return a result as true or false, depending on the comparison conditions.

Relational operators are used to establish some sort of relationship between the two operands. Some of the relevant examples could be less than (<), greater than (>), greater than or equal to (>=) operators. Python can interpret these types of operators and return appropriate results.

OPERATOR	MEANING	SYNTAX
<	Less than: True if the left operand is less than the right	`num1<num2`
>	Greater than: True if the left operand is greater than the right	`num1>num2`
= =	Equal to: True if both operands are equal	`num1= =num2`
!=	Not equal to: True if operands are not equal	`num1!=num2`
>=	Greater than or equal to: True if left operand is greater than or equal to the right	`num1>=num2`
<=	Less than or equal to: Returns True if the left operand is less than or equal to the right operand	`num1<=num2`

List of relational operators

Take the following variables with values as listed:

```
num1=5; num2=9
varA='cat'
varB='cot'
```

Here is a sample of the code with all these operators in use:

```
#using relational operators
num1=5; num2=9
varA='cat'
varB='cot'
print(num1>num2) #Greater than=>Returns False because num1
is less
print(varA<varB) #Greater or less can be used on relational
operators =Here returns Tru because 'a' starts before 'o'
print(num1==num2)#Equal returns false
print(4=='4') #returns false because number 4 and string4
are not equal to each other
print(num1!=num2) #Not equal to=> returns true
print(num1>=num2) #returns true because 5 is less than 9
print("abc" > "aaa") #returns True
print("abc" == "bcd") #returns false because the strings are
not the same
print(True>False) #Returns True because True is a 1 and
False a 0
```

In the line `print(num1 > =num2)`, the condition *Greater than* is `False` and the condition *Equal* to is `False`. If condition 1 and condition 2 are false, the result is false. In computer science, these various combinations are usually presented in the form of a table called the *truth table*, shown in the table below.

CONDITION –1	CONDITION –2	OR: TRUTH VALUE
False	False	False
False	True	True
True	False	True
True	True	True

Table 8.1: Truth table

8.2.2 Relational expression containing both numeric variables and literals

In this section, we are going to implement relational operators containing both numeric and literals. Let us consider the following challenge.

EXAMPLE 8.1

Write a program that can be used by the university to inform users whether they qualify for entry through reduced points. If a prospective applicant is older than or equal to 18 years but younger than 25 and is from a previously disadvantaged race, he/she qualifies for entry through reduced points.

EXAMPLE 8.1 (CONTINUED)

The output of the program must be:

```
Please enter your firstname: Sibongile
Please enter age: 19
Please enter race: African
Sibongile You qualify for reduced points entry for university
>>>
```

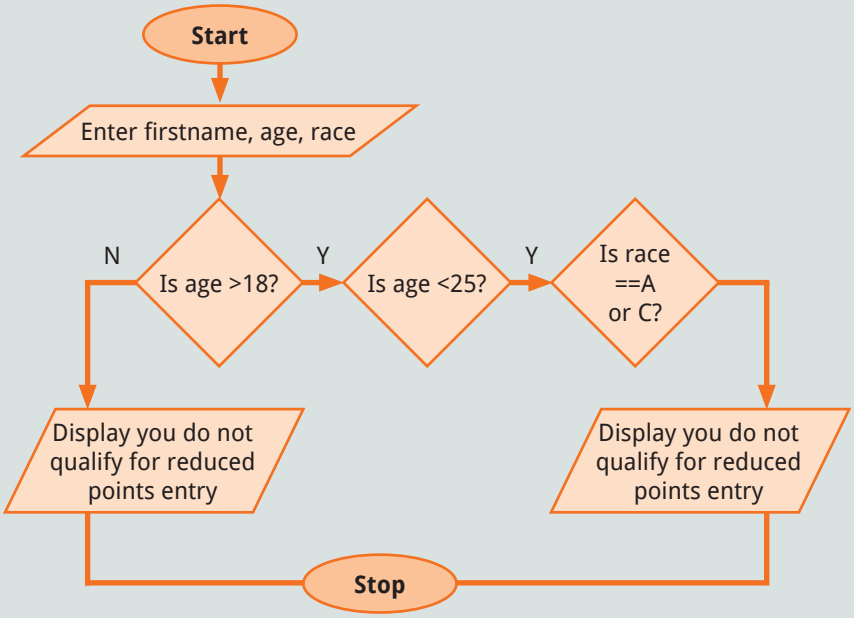
DID YOU KNOW

Some university departments have an affirmative-action policy that caters to previously disadvantaged communities. Affirmative action is the active effort to improve employment, educational and other opportunities for members of groups that have been subjected to discrimination.

Solution

1. **Planning**

   For this program, let us use a flow chart to depict the flow of events.



Flow chart for university entry

2. **Implementation of solution in Python**

   The following solution should be implemented:

```
try:
  #Accepting input
  firstname=input("Please enter your firstname: \t")
  age=int(input("Please enter age :\t"))
  race=input("Please enter race : \t")[0]
except:
  print("Invalid entries. Please re-enter the correct
  input")
else:
  print("All input passed the checks")
  #checking condition using relational operators
  if age>=18 and age<25 and race.upper( )=='A' or
  race.upper( )=='C':
    print(f'{firstname} \t You qualify for reduced
    points entry for university')
  else:
    print(f'{firstname}\t You don't qualify for
    reduced points entry for university')
```

In the program above, the input is placed after the *try* block. If an error occurs, the program jumps to the *except* block. If all the checks are completed and passed, the conditions are checked to see whether a student qualifies. You will also notice that the example on the previous page implemented not only comparison operators, but also logical operators, which are covered in the next section. Here is another challenge to strengthen your knowledge of selection structures.

EXAMPLE 8.2

A bank will offer loans to clients who are 21 or older, have an annual income of at least R21 000 and are permanently employed. The customers' age, permanency and income are inputs in response to user-friendly prompts. One of the following strings is printed, depending on the condition of an `if... else...` selection construct. Write a computer program to implement the following specification.

The output should be like this:

```
Enter the customer's age : 23
Enter the customer's annual income: 56320
Are you permanently employed [Yes or No] : y
We are able to offer you a loan.
>>>
```

Solution

```
try:
  #Capture the input
  customerAge=int(input("Enter the customer's age: \t"))
  annualIncome=int(input("Enter the customer's annual
  income: \t"))
  permanency=input("Are you permanently employed [Yes or
  No] :\t")[0]
except:
  print("Invalid entries")
else:
  if customerAge>=21 and annualIncome>=21000 and
  permanency.upper( )=='Y':
    print("We are able to offer you a loan.")
  else:
    print("Unfortunately at this time we are unable to
    offer \ you a loan.")
```

8.2.3 Logic operators in Python

A *logical operator* is a symbol or word used to connect two or more expressions such that the value of the compound expression produced depends only on that of the original expressions and on the meaning of the operator. Common logical operators include AND, OR and NOT.

The operands of a logical expression can be expressions that yield `True` or `False` when evaluated. There are three basic types of logical operators:

- Logical AND: For AND operations, the result is True only if both operands are True. The keyword used for this operator is *and*.
- Logical OR: For OR operations, the result is True if either of the operands is True. The keyword used for this operator is *or*.
- Logical NOT: The result is True if the operand is False. The keyword used for this operator is *not*.

```
#Logical operator examples
a = True; b = False
print(a and b) #False
print(a or b) #True
print(not b) #True
```

8.2.4 Python code that will save the result of relational expressions containing two or a mix of numeric variables, character variables, literals and multiple logical operators

There are times when the program needs to take in characters and numbers with multiple operators. *If-elif-else* can handle such complex conditional statements. Here is another challenge.

EXAMPLE 8.3

Write a Python program to accept the age, sex ('M', 'F') and number of days an employee has worked and display the wages accordingly.

Use the data provided in the table below:

AGE	SEX	WAGE/DAY
>=18 and <30	M	700
	F	750
>=30 and <=40	M	800
	F	850

If age does not fall in any range, then display the following message: "Enter appropriate age". Add a `try ... except` block to you program.

Solution

```
try:
  #capturing the input
  age=int(input("Enter your age \t"))
  sex=input("Enter sex(M/F) \t")
  num_of_days = int(input("Enter number of days \t"))
  #checking the condition
  if age >=18 and age < 30 and sex.upper( ) == 'M':
```

EXAMPLE 8.3 (CONTINUED)

```
    #Processing
    amt = num_of_days*700
    #displaying output
    print('Total wages is :R {:.2f}'.format(amt))
  elif age >=18 and age < 30 and sex.upper( ) == 'F':
    amt = num_of_days*750
    print('Total wages is :R {:.2f}'.format(amt))
  elif age >=30 and age <= 40 and sex.upper( ) == 'M':
    amt = num_of_days * 800
    print('Total wages is :R {:.2f}'.format(amt))
  elif age >=30 and age <= 40 and sex.upper( ) == 'F':
    amt = num_of_days * 850
    print('Total wages is :R {:.2f}'.format(amt))
  else:
    print("Enter appropriate age")
except:
  print("Invalid entries!!!. Restart again")
```

Practical activity 8.1 — INDIVIDUAL ACTIVITY

Write a program to accept two numbers and mathematical operators and perform operation accordingly.

For example: Enter first number: 7
Enter second number: 9
Enter operator: +
Your answer is: 16 (20)

TOTAL: 20

Activity 8.2 — INDIVIDUAL ACTIVITY

1. List THREE types of logical operators. (3)
2. Indicate whether the following fragments are valid or invalid first lines of *if* statements. Give a reason if your answer is 'invalid'.
   - **2.1** `if(x>4)` **2.2** `if x==2`
   - **2.3** `if(y=<4)` **2.4** `if(y=5)`
   - **2.5** `if(3<=a)` **2.6** `if(1-1)`
   - **2.7** `if((1-1)<=0)` **2.8** `if(name=="James")` (12)
3. List SIX types of relational operators. (6)
4. What is a correct Python expression for checking whether a number stored in a variable *x* is between 0 and 5?
   - **A** `x > 0` and `< 5`
   - **B** `x > 0` or `x < 5`
   - **C** `x > 0` and `x < 5`
   - **D** `x=0` to `5` (1)

TOTAL: 22

8.3 Selection statements

We have already identified the three programming constructs as sequential, selection or conditionals and iteration/looping/repetition. In the previous modules, you were introduced to the concept of flow of control: the sequence of statements that the computer executes. In procedurally written code, the computer usually executes instructions in the order in which they appear.

However, this is not always the case. One of the ways in which programmers can change the flow of control is by using selection statements. Selection statements are used in programming to select particular blocks of code to run based on a logical condition. In Python, the selection statements are also known as *decision making statements* or *branching statements*.

Python provides the following selection statements:

- *if* statement
- *if-else* statement
- *if-elif-else* statement.

8.3.1 *If* statement in Python

In Python, we use the *if* statement to test a condition and decide whether or not to execute a block of statements based on the outcome of that condition. If the condition is true, the statement block is executed; if it is false, the statement block is ignored. The execution path of *if* statements is illustrated below:

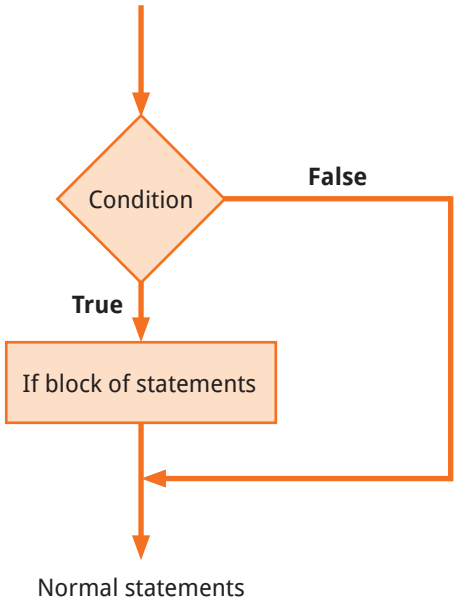


Figure 8.2: Flow chart of* if *statement

The general syntax of *if* statements in Python is:

```
If condition:
  Statement 1
  Statement 2
  Statement 3
```

When we define an *if* statement, the block of statements must be specified using indentation only. Look at the following example. Two numbers are captured through the keyboard and compared to tell the user which one is greater.

```
#enter the two numbers
num1=int(input("Please enter first number"))
num2=int(input("Please enter the second number"))
#comparison
if (num1>num2):
  print(f'{num1} \tis greater than \t {num2}')

#Output
Please enter first number56
Please enter the second number41
56 is greater than 41
>>>
```

The last line of the output is only printed if num1 is greater. In this instance, the output is printed only when the condition is `True`. What happens if the condition is `False`, e.g. if the value of num2 is greater? Nothing is printed out. In such cases, the *if* block will have to be extended and an *else* segment added, as explained in the following paragraph.

The *if-else* statement

We can use the *else* statement with an *if* statement to execute a block of code when the condition is false.

Syntax:

```
if (condition):
  #Executes this block if
  #condition is true
else:
  #Executes this block if
  #condition is false
```

Here is the flow chart for the *if-else* statement:

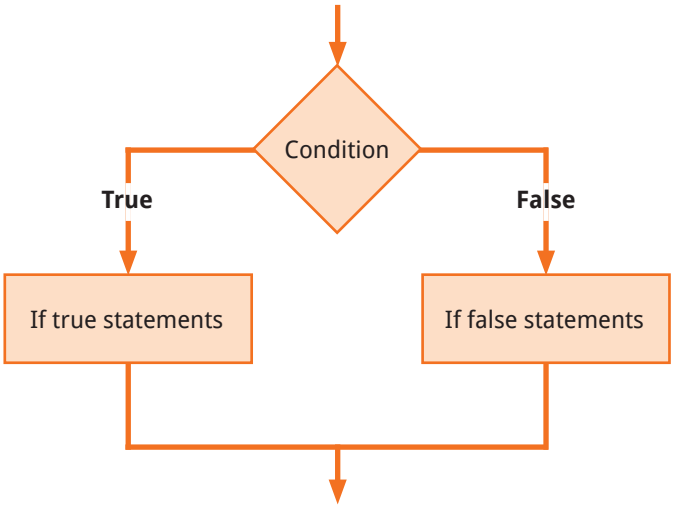


Figure 8.3: Flow chart for* if-else *statement

To accommodate the false condition when comparing two numbers, the code will be the following:

```
#enter the two numbers
num1=int(input("Please enter first number"))
num2=int(input("Please enter the second number"))
#comparison
if (num1>num2):
  print(f'{num1} \tis greater than \t {num2}')
#to be executed if the first condition is false
else:
  print(f'{num2} \tis greater than \t {num1}')

#Output
Please enter first number56
Please enter the second number75
75 is greater than 56
>>>
```

Given the above example, you would have noticed that the program is now accommodating the execution of statements if the first condition is false.

But you might wonder: What happens if the user enters two numbers that are of the same value, e.g. num1 = 23 and num2 = 23? By default, the last statement would be printed out. To avoid this, Python uses the *elif* statement, which is closed with an *else* statements as explained next.

The *if-elif-else* statement

Here, a user can choose among multiple options. The *if* statements are executed from the top down. As soon as one of the conditions controlling the *if* is true, the statement associated with that *if* is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final *else* statement will be executed.

To further your understanding, the program comparing the two numbers used previously has been rewritten and implemented using the *elif-else* block.

Syntax:

```
if (condition):
  statement
elif (condition):
  statement

else:
  statement

See the code sample below
#enter the two numbers
num1=int(input("Please enter first number"))
num2=int(input("Please enter the second number"))
#comparison
if (num1>num2):
  print(f'{num1} \tis greater than \t {num2}')
```

```
#to be executed if the first condition is false
elif (num1<num2):
  print(f'{num2} \tis greater than \t {num1}')
else:
  print( print(f'{num2} \tis equal to \t {num1}')

#Output
Please enter first number23
Please enter the second number23
23 is equal to 23
>>>
```

Now our code accommodates all three possibilities. If a condition is satisfied, we can have more than one statement executed. In the examples given in this section so far, only one statement has appeared in the *if* body.

VOCABULARY

Indentation – *spaces at the beginning of a code line*

Block – *piece of Python program text that is executed as a unit*

Of course, it is possible to have more than one statement following the *if* body; for example, when you want to evaluate a condition and then execute more than one option should the condition prove true. This is where grouping of statements, **indentation** and **blocks** come in handy.

Remember, instructions written in the source code for execution are called *statements*. A block is a combination of all these statements. Blocks can be regarded as a grouping of statements for a specific purpose. Statements that should be executed together will be indented the same under each block. The interpreter will treat all the statements inside the indented block as one statement – it will process all the instructions in the block before moving on to the next instruction. This allows us to specify multiple instructions to be executed when the condition is met.

Here you have an example of a grouping of statements, indentation and a block:

> If the weather is nice, then I will:
>
> - Mow the lawn
> - Weed the garden
> - Take the dog for a walk
>
> (If the weather isn't nice, then I won't do any of these things.)

DID YOU KNOW

We can write the `if-else` *statement in one line, as shown in line 5 below:*

```
1  #enter the two numbers
2  num1=int(input("Please enter first number"))
3  num2=int(input("Please enter the second number"))
4  #comparison
5  print(f'{num1}\tis greater than\t {num2}')if (num1>num2)
   else print(f'{num2} \tis greater than \t {num1}')
```

Activity 8.3 **INDIVIDUAL ACTIVITY**

Write a Python program to check whether a given year is a leap year or not. The program must ask the user to enter a year, then check and print the message whether it is a leap year or not.

Condition for leap year

A year can be called a *leap year*, only if:

- it is divisible by 4, but not by 100; or
- it is divisible by 400. (15)

TOTAL: 15

8.3.2 Relational expressions and logic operators in *if-else* statements

A good example of implementing relational and logic operators in the same program would be to solve a grading system, which can be used by educational institutions for student marks.

EXAMPLE 8.4

Write a program to enter five test scores of a subject and calculate the average mark. The program should display the grade.

Solution

- Take the student's name.
- Take in the marks of the five subjects of the user and store them in different variables.
- Find the average of the marks.
- Use an *else* condition to decide the grade based on the average of the marks.
- Exit.

Grade will be calculated according to:

1. `score >= 90: "A"`
2. `score >= 80: "B"`
3. `score >= 70: "C"`
4. `score >= 60: "D"`
5. `score >= 50: "E"`
6. `score < 50: "F"`

Program/Source code

Here is the source code of the Python program to take in the marks of five subjects and display the grade. The program output is also shown below:

```
#Grading system
#enter student name
studentName=input("Please enter the student name \t")
#enter the five student marks
mark1=int(input("Please enter the first test score \t"))
mark2=int(input("Please enter the second test score \t"))
mark3=int(input("Please enter the third test score \t"))
mark4=int(input("Please enter the fourth test score \t"))
mark5=int(input("Please enter the fifth test score \t"))
```

EXAMPLE 8.4 (CONTINUED)

```
#calculate average mark
totalScore=mark1 +mark2 +mark3 +mark4 +mark5
averageMark=totalScore/5
print(f'{studentName} your total score is {totalScore}')
print(f'Your average score is {averageMark}%')
if (averageMark>=90) and (averageMark<=100):
  print(f'{studentName} you got an A grade')
elif (averageMark>=80) and (averageMark<90):
  print(f'{studentName} you got an B grade')
elif (averageMark>=70) and (averageMark<80):
  print(f'{studentName} you got an C grade')
elif (averageMark>=60) and (averageMark<70):
  print(f'{studentName} you got an D grade')
elif (averageMark>=50) and (averageMark<60):
  print(f'{studentName} you got an E grade')
elif (averageMark>=0) and (averageMark<50):
  print(f'{studentName} you got an F grade')
else:
  print(f'{studentName} your mark is outside the range')
```

The above example implemented the >= relational operator and the *and* logical operator. The next example tests the usage of the comparison operator in *if* statements.

Activity 8.4 **INDIVIDUAL ACTIVITY**

Write a Python program to check whether a given number entered by a user is an even or an odd number. (10)

TOTAL: 10

8.3.3 The *nested if* statement

A *nested if* statement is an *if* statement that is the target of another *if* statement, i.e. one *if* statement is placed inside another *if* statement. Nesting levels can only be determined by indentation. Unless it is absolutely necessary, the use of *nested if* statements should be avoided.

Syntax:

```
if (condition1):
  #Executes when condition1 is true
  if (condition2):
    #Executes when condition2 is true
 #if Block is end here
#if Block is end here
```

EXAMPLE 8.5

Write a program to check whether a number is positive, negative or zero and display an appropriate message.

```
#enter a number of your choice negative or positive or 0
num = float(input("Enter a number: "))
if num >= 0:
  if num == 0:
    print("Zero")
  else:
    print("Positive number")
else:
  print("Negative number")

#Sample Output
Enter a number: -55
Negative number
Enter a number: 65
Positive number
Enter a number: 0
Zero
>>>
```

8.3.4 Using relational expressions in *nested if-else* statements

In this section, you are going to learn to write a Python program that uses relational expressions in *nested if-else* statements.

EXAMPLE 8.6

You are required to develop a shipping calculator for a company. The shipping charges for a package are shown below.

- 2 kilograms or less = R2,10
- over 2 kilograms but not more than 6 kilograms = R3,20
- over 6 kilograms but not more than 10 kilograms = R4,70
- over 10 kilograms = R4,80

The company does not offer services if the weight exceeds 100 kilograms.

Write a program that determines the total shipping costs and informs the user in accordance with the weight per item. The program must ask the user to enter the weight of a package and it must then display the shipping charge.

EXAMPLE 8.6 (CONTINUED)

Solution

Take note that we are revising the `try...except` block, which is used for handling errors.

```
try:
  #Accepting user input
  weight=float(input("Enter the amount of kilograms to be
  shipped \t"))
  #Checking the weight of the parcel and appropriate
  if (weight<2):
    print("The shipping cost is R", 2.10)
  elif (weight>2) and (weight <=6):
    print("The shipping cost is R", 3.20)
  elif (weight>6) and (weight<=10):
    print("The shipping cost is R", 4.70)
  elif(weight>10) and (weight<100):
    print("The shipping cost is R", 4.80)
  else:
    print("We are not able to assist as your load exceeds
the maximum limit")
except:
  #exception messages.
  print("Invalid entry. Try again")
```

Now that you have mastered conditional statements, you are going to program your Raspberry Pi so that when a certain condition is `True` or `False`, the hardware will respond. Complete the activity that follows.

Practical activity 8.2 **INDIVIDUAL ACTIVITY**

(Physical computing)

You are required to modify the shipping program above to include physical computing. You must do the following:

1. Connect and code a green LED light to turn on when the shipping weight is within the expected limit.
2. Connect and code a red LED to turn on for a few seconds if the package exceeds the limit of 100 kg.
3. The red LED must also turn on if a user enters the incorrect input. (20)

TOTAL: 20

8.3.5 The general form for a Python *if-else* chain

The syntax of an *if-else* statement is given:

```
if test expression:
  Body of if
Else:
  Body of else
```

As explained earlier, the *if-else* statement evaluates the test expression and executes the body only if the test condition is `True`; otherwise, the body of *else* is executed.

8.3.6 The *if-else* chain

You may have noticed that our previous program for grading students still has some shortfalls; for instance, it goes to print the average mark even though it is outside the range. We can revise the code and add an *if* block within the first *if* segment to check the range. If the range criteria are not met, no comparison will be made. Instead, the program will jump to the *except* block, as shown below:

```
#Grading system
try:
  #enter student name
  studentName=input("Please enter the student name \t")
  #enter the five student marks
  mark1=int(input("Please enter the first test score \t"))
  mark2=int(input("Please enter the second test score \t"))
  mark3=int(input("Please enter the third test score \t"))
  mark4=int(input("Please enter the fourth test score \t"))
  mark5=int(input("Please enter the fifth test score \t"))
  #calculate average mark
  totalScore=mark1 +mark2 +mark3 +mark4 +mark5
  averageMark=totalScore/5
  if (averageMark>100) or (averageMark<0):
    print('some of your entries are outside the range')
  else:
    print(f'{studentName} your total score is {totalScore}')
    print(f'Your average score is {averageMark}%')
    if (averageMark>=90) and (averageMark<=100):
      print(f'{studentName} you got an A grade')
    elif (averageMark>=80) and (averageMark<90):
      print(f'{studentName} you got an B grade')
    elif (averageMark>=70) and (averageMark<80):
      print(f'{studentName} you got an C grade')
    elif (averageMark>=60) and (averageMark<70):
      print(f'{studentName} you got an D grade')
    elif (averageMark>=50) and (averageMark<60):
      print(f'{studentName} you got an E grade')
    else:
      print(f'{studentName} you got an F grade')
except:
  print("Please re-enter the input")
```

Comparison by *is* and ==

A common pitfall in Python is confusing the equality comparison operators *is* and ==. So far, we have only compared integers in our examples. We can also use any of the above relational operators to compare floating-point numbers, strings and many other data types:

```
#we can compare the values of strings
if name == "Siyanda":
  print("Hello, Siyanda!")

#... or floats
if size < 20.8:
  print(size)
```

VOCABULARY

Identity (of object) – *unique, constant integer that exists for the length of the object's life*

When comparing variables using ==, we are doing a *value comparison*; we are checking whether the two variables have the same value. In contrast to this, we might want to know whether two objects, such as lists, dictionaries or custom objects that we have created ourselves, are the exact same object. This is a test of **identity**. Two objects might have identical content, but be two different objects.

We compare identity using the *is* operator:

- `a == b` compares the value of *a* and *b*.
- `a is b` will compare the identities of *a* and *b*.

The concept is illustrated with some code statements below:

```
>>> a='Python is great'
>>> b='Python is great'
>>> a==b #returns true
True
>>> a is b #returns false because a is not b
False
>>>
>>> x=55
>>> z=x
>>> x==z
True
>>> x is z
True
>>>
```

The == operator compares the value or *equality* of two objects, whereas the Python *is* operator checks whether two variables point to the same object in memory.

Conditional statements regulate the control flow of a Python program.

There are different types of conditional statements: *if*, *if-else*, *elif*, *nested if* and *nested if-else* statements. If a condition is `True`, then the statement inside the *if* block will be executed; if `False`, the statements in the *else* block will be executed. If there are multiple conditions, Python uses *elif* to test them, and whichever equates to true will be executed.

The next programming construct after selection structures is repetition structures, which will be discussed in the following module.

Activity 8.5 **INDIVIDUAL ACTIVITY**

1. List the THREE different Python statements used for *selection constructs*. (3)
2. The condition in the *if* statement should be in the form of a/an ...
   - **A** arithmetic or relational expression.
   - **B** arithmetic or logical expression.
   - **C** relational or logical expression.
   - **D** arithmetic and assignment. (1)
3. An *if* condition inside another *if* is called ...
   - **A** the second *if*.
   - **B** a *nested if*.
   - **C** another *if*.
   - **D** None of the above. (1)
4. Leading whitespace (spaces and tabs) at the beginning of a statement is called ...
   - **A** indentation.
   - **B** orientation.
   - **C** iteration.
   - **D** None of the above. (1)
5. What is the output of the code snippet shown below? (2)

```
if True:
 print("Hello")
else:
 print("Bye")
print("enjoy")
```

6. Which symbol is used to terminate an *if* statement in Python? (1)
7. Consider the following *if-else* statement to compare the two numbers:

```
x=10
y=7
if x>y:
  print(y, “is less than”, x)
else:
  print(x, “is less than”, y)
```

   Rewrite the above statement using *if* statement shorthand. (5)
8. Write a Python program to accept the length and breadth values of a shape from the user and check whether the shape is a square or not. (6)
9. What are *elif* statements in Python? (2)
10. You are tasked by the Johannesburg Metro Police to write a program for their new Camera Driver Demerit System. The program should ask the driver to give his/her current speed in km/h and the average allowed speed of the road. The program should conform to the following requirements:
    - If the speed is less than 60 km/h, it should print `OK`.

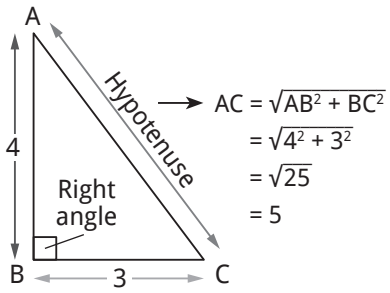
Activity 8.5 (continued)

- Otherwise, for every 5 km above the speed limit (i.e. 60 km/h), it should give the driver one demerit point and print the total number of demerit points. For example, if the speed is 70 km/h and the average speed in that area is 60 km/h, it should print: *Demerit Points: 2*.
- If the driver gets more than 12 demerit points, the function should print *Time to go to jail!* (15)

TOTAL: 37

Summative assessment

1. Write a Python program to determine the third side of a right-angled triangle based on the two given sides. Your program should allow the user to input the size of two sides and then calculate the other side. (20)



2. What would the following code print?

```
a = 6
b = 10
print (not (not a==10 or not b ==10))
```
(1)

3. Explain the syntax of an *if-else* statement and state when you would apply the statement. (5)

4. For which values of input will this program print `True`?

```
if not input > 5:
  print("True")
```
(2)

5. For which values of *absentee_rate* and *overall_mark* will this program print "You have passed the course."?

```
if absentee_rate <= 5 and overall_mark >= 50:
  print("You have passed the course.")
```
(2)

TOTAL: 30

NOTE

In mathematics, the Pythagorean theorem (also known as Pythagoras' theorem) is a fundamental relation in Euclidean geometry among the three sides of a right triangle. It states that the square of the hypotenuse (the side opposite the right angle) is equal to the sum of the squares of the other two sides.

Module 9

Repetition control structure

After you have completed this module, you should be able to:

- write Python code that uses relational expressions and logic operators containing a *while* loop;
- write Python code that uses relational expressions containing logic operator in *while* loops with compounded content;
- explain what the following statements are used for in the Python *while* loop:
  - *break* statement
  - *pass* statement
  - *continue* statement;
- write Python code that incorporates the *break*, *continue* and *pass* statements;
- identify or correct the general form for a *for* statement using the range function;
- explain the purpose of each part in the `for` loop initialisation using the range function;
- use the *break* statement to end an infinite *for* loop based on the loop reaching a condition;
- write Python code that will use a *for* loop with:
  - a predetermined number of loops
  - a non-sequential counter variable as defined by the range function;
- define the term *nested loop*;
- define the terms *inner* and *outer loops*;
- write Python code that will nest:
  - identical loops
  - different loops; and
- determine the application flow when a nested loop statement is encountered.

Introduction

After revising the basic concepts of visual programming and the skills you have learned until now, we will focus on the loop function used in Python, more specifically the *while loop* and *nested loops*.

9.1 Revision of concepts covered in previous modules

In Module 3, you were introduced to the concept of visual programming. A VPL is a computer program that develops applications using graphical components and figures. Scratch is one example of a VPL. We also covered concepts such as *variables*, *data types*, *operators* and *programming constructs*.

By now you are aware that there are three programming structures, and these are:

- sequence;
- selection; and
- looping or iteration structures.

Sequence and selection structures have been implemented in problem solving using Python and Scratch. The use of loops in Scratch was introduced in Module 3. To refresh your memory, you will be tasked with a problem that must be solved using the looping construct in Scratch.

In general, statements are executed sequentially; the first statement in a function is executed first, followed by the second, and so on. There may be situations where you need to execute a block of code several times. Programming languages provide various control structures that allow for more complicated execution paths. In a programming language, a *loop* is a statement that contains instructions that are continually repeated until a certain condition is reached. In other words, a loop statement allows us to execute a statement or group of statements multiple times. The figure below illustrates a loop statement. Take note that the statements in the loop body will continue to be executed for as long as the condition holds; once it turns to false or is not satisfied, the program will terminate.

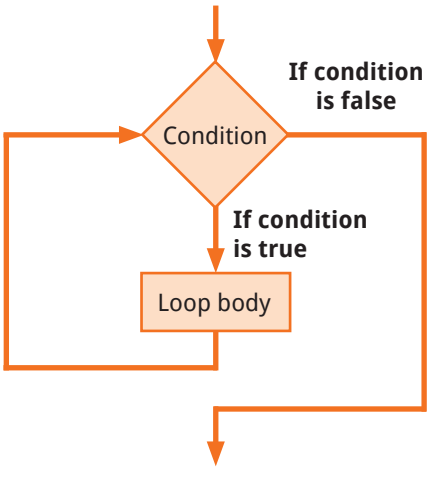


Figure 9.1: Loop structure

In Python, looping is handled by the following types of loops:

- `while ...` loop
- `for ...` loop
- `nested` loops

Importance of looping

Loops simplify complex problems by altering the flow of the program so that, instead of writing the same code repeatedly, we can repeat it for a finite number of times. With the use of loops, we can cut those hundred lines of code down to a few lines. We can, for example, print the first 10 natural numbers by using a loop instead of using the print statement 10 times.

Advantages of loops

Loops have the following advantages:

- Code can be reused
- Redundancy is removed
- Solutions can be developed more quickly compared to sequence structures
- They enable us to traverse data structures (arrays or linked lists) easily.

Practical activity 9.1 **INDIVIDUAL ACTIVITY**

Using the knowledge of Turtle programming gained in Module 3, write a Python program using Turtle to draw a hexagon and fill it in with a colour of your choice. The program must ask a user to enter the length of the side and the colour. (10)

9.2 *While* repetition control

9.2.1 Writing code using relational expressions and logic operators

A *while* loop statement in the Python programming language repeatedly executes a target statement for as long as a given condition is true. For instance, you may want to repeat saying "Hello, Good Morning" for as long as it is still morning when you meet someone. So, while it is still morning, you keep repeating the same greeting.

Syntax of a *while* loop:

```
while expression:
  statement(s)
```

Figure 9.1 (provided earlier) illustrates the implementation of the while loop. It is important to note that the statement(s) may be a single statement or a block of statements. The *while* loop is an example of a pre-test looping structure. This means that the condition is checked first before the execution of any statements. Only if the condition is true will the statements be executed; otherwise, the program code will jump to execute instructions to

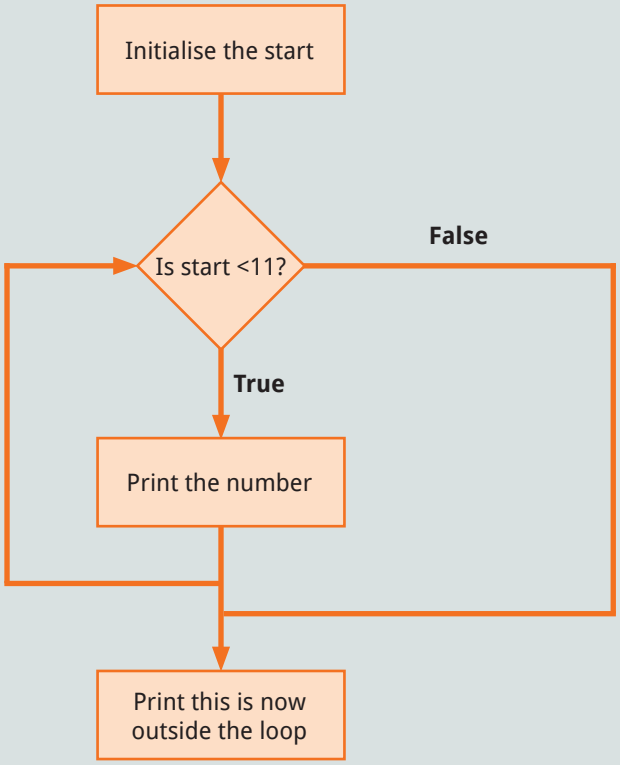
be done when the loop is false. So, if the condition is not met given the first execution statements in the loop body, it might never be executed at all. In light of the previous example, if time is always afternoon, the "Hello, Good Morning" statement would never be executed at all.

Look at the following example of a *while* loop.

EXAMPLE 9.1

Write a program to print all integers from 1 to 10 using a *while* loop. Use PyCharm IDE.

Planning:
Flow chart to illustrate the scenario.



The* while *loop

Solution

```
#using a while loop
#set the starting point
start=1
#add the terminating condition
while start <11:
  #display the numbers
  print(f'Number {start} is :' , start)
  #increment the counter
  start=start+1
#code outside the loop
print("Now out of the loop")
```

EXAMPLE 9.1 (CONTINUED)

```
#Output
Number 1 is : 1
Number 2 is : 2
Number 3 is : 3
Number 4 is : 4
Number 5 is : 5
Number 6 is : 6
Number 7 is : 7
Number 8 is : 8
Number 9 is : 9
Number 10 is : 10
Now out of the loop
```

We have managed to use Python to solve a problem that required looping and we implemented the *while* loop.

Here is another example to further your understanding of this loop:

EXAMPLE 9.2

Write a program that uses a *while* loop and print all multiples of 5 from 50 to 0.

```
#multiples of 5
#initialise the divisor
multiple=5
#initialise the starting position
start=50
#add the condition
while start>0:
  #check for modulus of 5the number
  if start%multiple==0:
    print(start)
  #decrement the value of the number
  start=start-1

Output
50
45
40
35
30
25
20
15
10
5
```

Using *else* statement with *while* loops

As discussed above, the *while* loop executes the block until a condition is satisfied. When the condition becomes false, the statement is executed immediately after the loop is completed.

NOTE

The **else** *clause is only executed when your* **while** *condition becomes false. If you break out of the loop or if an exception is raised, it will not be executed. As explained in section 5.1.2, an exception is an error that happens during the execution of a program.*

Using the previous example of multiples of 5, if we want to add the *else* block, the code will look as follows:

```
start=50
#add the condition
multiple=5
while start>0:
  #check for modulus of 5the number
  if start%multiple==0:
    print(start)
  #decrement the value of the number
  start=start-1
else:
  print("The loop has terminated")

Output
50
45
40
35
30
25
20
15
10
5
The loop has terminated
```

If we change the value of start to 0, the statement in the *else* block will be executed because the program starts with a false condition.

9.2.2 Relational expressions containing logic operators in *while* loop

A more challenging task is implementing multiple logical operators and *if* statements in the *while* loop. In some instances, loops must solve complex problems, which may require the use of selection and sequence structures with numerous conditions. You will be required to use logical and relational operators to evaluate the conditions for the loop to execute or not to execute.

Consider the following scenario:

EXAMPLE 9.3

You are requested to write a Python program that iterates the integers from 20 down to 1. For multiples of three, the program must print "Fizz" instead of the number, and for multiples of five it must print "Buzz". For numbers that are multiples of both three and five, it must print "FizzBuzz". For numbers that are not multiples of both 3 and or 5, it must print the actual number.

Solution

```
#initialise the counter to 20 as the starting point
count = 20
#condition
while count > 0:
  #statements to be executed if condition is true
  if ((count % 5) == 0 and (count % 3) == 0):
    #displaying output
    print("FizzBuzz")
    #reducing the value of the counter after every
    iteration
    count = count - 1
  elif (count % 3) == 0:
    print("Fizz")
    count = count - 1

  elif (count % 5) == 0:
    print("Buzz")
    count = count - 1
  else:
    print(count)
    count = count – 1

Output
Buzz 19 Fizz 17 16 FizzBuzz 14 13 Fizz 11 Buzz Fizz 8 7
Fizz Buzz 4 Fizz 2 1
```

NOTE

To print output in the script area in a horizontal line instead of one line after the other, Python uses the string `end = =  =`.

As you will notice in the output, instead of 20, the program printed *Buzz* because 20 is divisible by 5 only. For 19, the value was printed because 19 leaves a remainder when divided by both 3 and 5. Instead of 15, the program printed the word *FizzBuzz* because 15 is divisible by both 3 and 5.

You might face a situation in which you need to exit a loop completely when an external condition is triggered, or there may also be a situation where you want to skip a part of the loop and start the next execution. This is where control statements in Python come in handy. Control statements are used to control the flow of execution of the program based on a specified condition.

Python supports three types of control statements:

- Break
- Continue
- Pass.

These statements will be discussed in the sections that follow.

9.2.3 Using Python *break* statements in the *while* loop

The *break* statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop. If the *break* statement is inside a *nested* loop (a loop inside another loop), the *break* statement will terminate the innermost loop.

Example

```
i = 0
while i < 7:
  print(I, end =" ")
  if i == 4:
    print("Breaking from loop")
    break
  i += 1

Output
0 1 2 3 4 Breaking from loop
```

In this example, the condition is that if the value of *i* is exactly equal to 4, the program must terminate. However, take note that 4 is printed, as the printing of *i* happens before the break condition.

Here is another example of implementing a *break* statement.

EXAMPLE 9.4

Write a Python program to print multiples of 3 from 99 to 3. Invoke a *break* statement to terminate the loop as soon as the counter is below or equal to 50.

The printout should be as follows:

```
99 96 93 90 87 84 81 78 75 72 69 66 63 60 57 54 51
```

Solution

```
#initialising the counter
count=99
#condition
while count>=3:
  #selection statement to break when counter is 50
  if (count <= 50):
    break
  #statement to be run if selection statement is false
  if count%3==0:
    print(count, end= " ")
  count=count-1
```

9.2.4 *Pass* statement used in Python *while* loop

The *pass* statement is a null statement that can be used when a statement is required by Python syntax, but no action is required or desired by the programmer (such as within the body of a *for* or *while* loop). This can be useful as a placeholder for code that is yet to be written.

```
while x == y:
  pass
```

In this example, nothing will happen. The *while* loop will complete without error, but no commands or code will be actioned. Using *pass* allows us to run our code successfully without having all commands and action fully implemented.

For the purposes of practice, modify the code below:

```
i = 0
while i < 7:
  print(I, end =" ")
  if i == 4:
    print("Breaking from loop")
    break
  i += 1
```

Replace the *break* statement with a *pass* statement, as shown below:

```
i = 0
while i < 7:
  print(i, end =" ")
  if i == 4:
    print("Breaking from loop")
    pass
  i += 1
```

Activity 9.1 **INDIVIDUAL ACTIVITY**

Discuss the effect of the *pass* statement.

Similarly, *pass* can be used in *for* loops, as well as in selections and function definitions, etc.

```
for x in range(10):
  pass
```

9.2.5 Using the *continue* statement with the *while* loop

The *continue* statement will skip to the next iteration of the loop, bypassing the rest of the current block but continuing the loop. As with *break*, *continue* can only appear inside loops. The working of the *continue* statement in the *while* loop is shown on the next page.

```
while test expression:
    #codes inside while loop
    if condition:
        continue
    #codes inside while loop
#codes outside while loop
```

EXAMPLE 9.5

Write a Python program using a *while* loop and *continue* statement to print numbers from 10 to 1. The program should skip the value 5.

```
#initialise counter
count = 10
#set the condition
while count > 0:
  #decrement the counter
  count = count - 1
  #condition to be skipped
  if count == 5:
    continue
  #what to be printed apart in the loop
  print('Current variable is :', count)
print("Out of the loop-exiting the program!")
```

The output of the program will be as follows:

```
Current variable is : 9
Current variable is : 8
Current variable is : 7
Current variable is : 6
Current variable is : 4
Current variable is : 3
Current variable is : 2
Current variable is : 1
Current variable is : 0
Out of the loop-exiting the program!
```

Practical activity 9.2 **INDIVIDUAL ACTIVITY**

Write a Python *while* loop to print multiples of 3 from 100 to 0. Implement the *continue* statement to skip value 48 and all multiples of 3 after printing 21.

```
99 96 93 90 87 84 81 78 75 72 69 66 63 60 57 54 51 45
42 39 36 33 30 27 24 21 18 15 12 9 6 3 Good bye!
```

(10)

TOTAL: 10

9.2.6 Using the *break*, *continue* and *pass* statements

It is possible to implement *break*, *continue* and *pass* statements in one program. You might want to write a program that skips if a certain event occurs by using *continue*; executes even if certain conditions happen by implementing a *pass* statement; and exits the loop when a certain condition occurs by executing a *break* statement.

EXAMPLE 9.6

Write a Python program that loops from 100 to 5 and prints multiples of 5 using a *while* loop. The program must skip 60 and pass when the value is 55. As soon as the program counter reaches 50, the program must exit the loop and print the text "Good bye". Make sure you use *break*, *continue* and *pass* statements within the loop.

The output should look as follows:

```
95, 90, 85, 80, 75, 70, 65, 55, Good bye!
```

Solution

```
#counter
counter = 100
#condition
while counter > 5:
  #decrementing the counter
  counter = counter - 1
  #selection control to check for multiples of 3
  if counter % 5 == 0:
    #condition to skip the value 60
    if counter == 60:
      continue
    #statements to execute when counter = 55
    if counter == 55:
      pass
    #terminating the loop when counter = 50
    if counter == 50:
      break
    print(counter, end=", ")
#the statement outside the loop
print("Good bye!")
```

In Module 6, we listed different data types, one of which was a list.

Lists are used to store multiple items in a single variable. Lists are created using square brackets. Here is an example of a list declaration to store five names of students:

```
student_names=['Tendai', 'Simon', 'Yolanda', 'Muhahid',
'Joseph']
```

Python uses index numbers to identify an item in a list. The first item has index [0], the second item index [1], etc. Study the following code snippet in Python:

```
student_names=['Tendai', 'Simon', 'Yolanda', 'Mujahid',
'Joseph']
print(student_names[3])

Output
Mujahid
```

Now we want to create the list using a *while* loop.

EXAMPLE 9.7

Write a Python program using a *while* loop and a *break* statement to print student names up to the third name "Yolanda".

Solution

```
student_names=['Tendai', 'Simon', 'Yolanda', 'Mujahid',
'Joseph']
i = 0
while True:
  print(student_names[i])
  if (student_names[i] == 'Yolanda'):
    print('The break point has been reached')
    break
    print('After break statement')
  i += 1
print('After a while-loop exit')
```

Modify the program to print Simon and Mujahid only.

```
student_names=['Tendai', 'Simon', 'Yolanda', 'Mujahid',
'Joseph']
i = len(student_names)
#initialise counter

#set the condition
while i > 0:
  #decrement the counter
  i = i - 1
  #condition to be skipped
  if i == 0 or i == 1 or i==4:
    continue
    if i==5:
      break
  #what to be printed apart in the loop
  print('Current name is :', student_names[i],'at
position', i)
print("Out of the loop-exiting the program!")
```

Activity 9.2 **INDIVIDUAL ACTIVITY**

1. Define the term *loop* as used in programming. (2)
2. List FOUR advantages of looping in programming. (4)
3. Consider the code below, which is aimed at producing the following output:

```
5
4
3
2
1

a=5
while a>0:
  print(a)
```

When the code above is executed, it keeps printing 5. Modify the code to print the initial expected output, as shown above. (2)

4. How many control statements are present in Python? (3)
5. What is the difference between a *break statement* and a *continue* statement in Python? (2)
6. What is the use of a *pass* keyword in Python? (2)
7. What will be the output of the following Python code? (2)

```
i = 1
while True:
  if i % 3 == 0:
    break
  print(i)
  i + = 1
```

A `1 2 3`
B `Error`
C `1 2`
D None of the above.

8. What will be the output of the following Python program? (1)

```
i = 0
while i < 5:
  print(i)
  i += 1
  if i == 3:
    break
else:
  print(0)
```

A `error`
B `0 1 2 0`
C `0 1 2`
D None of the above

9. It is possible to use *continue* in an *if* statement in Python? (1)

Activity 9.2 (continued)

10. What will the output of the following code be? (1)

```
True = False
while True:
  print(True)
  break
```

A False
B True
C Syntax error
D None

11. Fill in the blank so that the following Python code results in the formation of an inverted, equilateral triangle. (1)

```
import turtle
t=turtle.Pen( )
for i in range(0,3):
  t.forward(150)
  t.right(_____)
```

A –60
B 120
C –120
D 60

12. What will the output shape of the following Python code be? (1)

```
import turtle
t=turtle.Pen( )
for i in range(1,4):
  t.forward(60)
  t.left(90)
```

A Rectangle
B Trapezium
C Triangle
D Square

13. Write a Python program to accept a number and then add the sum of all the natural numbers up to the number entered by the user. (10)

```
sum = 1+2+3+4+…number

Example of printout:
Enter number of values : 6
2 3 4 5 6 7 The sum is 21
```

Example of printout:

```
Enter number of values : 6
2 3 4 5 6 7 The sum is 21
```

TOTAL: 32

9.3 The *for* repetition control

The *for* loops iterate over a collection of items, such as a list, dictionary, range, tuple, set, string or other iterable objects, and run a block of code with each element from the collection. An iterable is an object that can be looped over or iterated over with the help of a *for* loop. Iterating over a sequence is called *traversal*. Revisit the concept of data types in Module 6 and revise the different categories of data such as sequences and sets.

9.3.1 Using the range function

Syntax of *for* loop

```
for val in sequence:
  loop body
```

Here, *val* is the variable that takes the value of the item inside the sequence on each iteration. The loop continues until we reach the last item in the sequence. The body of the *for* loop is separated from the rest of the code using indentation. A flow chart depiction of a *for* loop is given below.

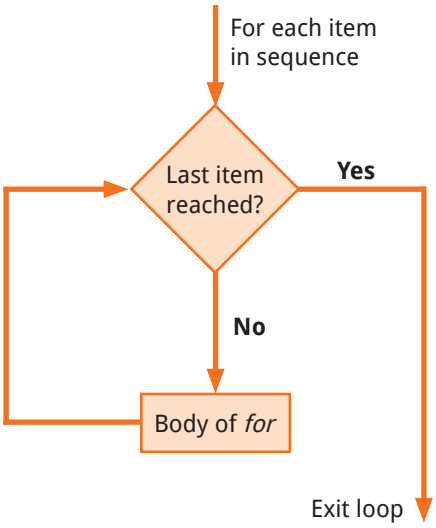


Figure 9.2: The* for *loop

Here we present an example of a *for* loop iterating through a list of cars:

```
#list of favourite cars
my_cars=['Ford', 'Toyota', 'Mercedes', 'VW', 'Suzuki']
for x in my_cars:
  print(x)
```

When the above code is executed, it produces the following result:

```
Ford
Toyota
Mercedes
VW
Suzuki
```

We can also iterate through a string and be able to print each letter. Strings are iterable objects, i.e. they contain a sequence of characters. To understand this, we are going to iterate through a string “Learning Python”.

```
for word in “Learning Python”:
  print(f’The current letter is {word}’)
```

When the above code is executed, it produces the following result:

```
The current letter is L
The current letter is e
The current letter is a
The current letter is r
The current letter is n
The current letter is i
The current letter is n
The current letter is g
The current letter is
The current letter is P
The current letter is y
The current letter is t
The current letter is h
The current letter is o
The current letter is n
```

You will notice that one of the lines just prints the text “The current letter is”, but there is no letter from the string “Learning Python”. This is because of the space between the two words “Learning” and “Python”. In section 9.3.3, we will introduce *break* and *continue* to control the loop, and we can skip whatever letter we do not want printed.

9.3.2 Purpose of each part in the *for* loop initialisation using the range function

If you do need to iterate over a sequence of numbers, the built-in function `range( )` comes in handy.

The `range( )` function returns a sequence of numbers, starting from 0 by default, incrementing by 1 (by default), and ending at a specified number. In Python, using a *for* loop with `range( )`, we can repeat an action a specific number of times. For example, let us see how to use the `range( )` function of Python 3 to produce the first five numbers. It generates arithmetic progressions.

Example

```
for i in range(5):
  print(i)
```

When the above code is executed, it produces the following result:

```
0 1 2 3 4
```

As you will notice in the printout, the given end point is never part of the generated sequence; `range(5)` generates 5 values, i.e. 0 to 4, the legal indices for items of a sequence of length 5. It is possible to have the range start at another number or to specify a different increment (even a negative one; sometimes this is called the 'step').

Syntax:
Below is the syntax of the `range( )` function:

```
range(start, stop[, step])
```

It takes three arguments. Of the three, two are optional. The *start* and *step* are optional arguments, and the *stop* is the mandatory argument.

Parameters

- `start:` (lower limit) The starting position of the sequence. The default value is 0 if not specified, e.g. `range(0, 5)`. Here, start = 0 and stop = 5.
- `stop:` (upper limit) Generates numbers up to this number, i.e. an integer number specifying at which position to stop (upper limit). As mentioned earlier, `range( )` never includes the stop number in its result.
- `step:` Specifies the increment value. The default value is 1 if not specified. It is simply a difference between each number in the result, e.g. `range(0, 5, 1)`. Here, step = 1.

We are going to write a Python program using a loop and range to print even numbers from 2 to 30.

```
for x in range(2, 30, 2):
  print(x, end =" ")

Output
2 4 6 8 10 12 14 16 18 20 22 24 26 28
```

In the above example, it is important to note that `for ... range` does not print the stop number. Also, the step 2 can be a negative value if you are doing a countdown loop, e.g. `range(30, 2, -2)`. This will start at 30 and decrement the loop with negative 2.

Iterating by sequence index

In addition to iterating over each item, you can also iterate over the sequence itself by *index offset*. Following is a simple example:

```
fruits = ['strawberry', 'banana', 'grape', 'peach', 'mango']
for index in range(len(fruits)):
  print(f'fruit number {index}:', fruits[index])
print('Finished printing all the fruits')

Output
fruit number 0: strawberry
fruit number 1: banana
fruit number 2: grape
fruit number 3: peach
fruit number 4: mango
Finished printing all the fruits
```

Practical activity 9.3 **INDIVIDUAL ACTIVITY**

Create a list of all the even numbers between 1 and 11. Print all the numbers and the sum of all the numbers.

The output should like as follows:

```
2 4 6 8 10
The sum of the even numbers between 1 and 11 is 30
```

(8)

TOTAL: 8

Loops can also be used to generate random numbers by making use of the `randint( )` inbuilt function. This function takes two arguments: the start and the end of the range for the generated integer values. Random integers are generated within and include the start and end of the range values, specifically in the interval [start, end]. Random values are drawn from a uniform distribution.

Here is a sample program to generate random numbers within a range. You will also recall the purpose of *import statement* if you want to make use of a module.

```
import random
n = random.randint(0,20)
print(n)
```

When the program is executed, a random number between 0 and 22 will be generated.

Practical activity 9.4 **INDIVIDUAL ACTIVITY**

Write a program that generates a random number (0–10) and asks you to guess it. You have three asserts.

- Define a random number between 0 and 10.
- Initialise `guesses_left` to 3.
- Use a *while* loop to let the user keep guessing for as long as `guesses_left` is greater than zero.
- Use an *else*: case after the *while* loop to print: "You lose". (15)

TOTAL: 15

9.3.3 Using the *break* statement to end an infinite *for* loop

We have identified three *control* statements that can change the execution of a loop as *control*, *break* and *pass*. The implementation is the same as in the *while* loop.

To illustrate this, we are going to write a Python program using a *for* loop to iterate from 100 to 1, printing multiples of 5. The program must skip value

65 and exit the loop if it encounters value 50. Make use of *break* and *continue* statements.

```
initialising the sum to 0
sum =0
#specifying start value and stop value
for i in range(100, 1,-1):
  #condition to test for even numbers
  if i % 5 == 0:
    #displaying output
    if i==65:
      continue
    if i<50:
      break
    print(f' {i}', end =" ")
    #Adding each even number to sum
    sum=sum+i
#displaying the sum
print(f'\nThe sum of the even numbers between 1 and 100 is
{sum}')

Output
100  95  90  85  80  75  70  60  55  50
The sum of the even numbers between 1 and 100 is 760
```

The code above demonstrates how you can use *break* and *continue* in the same program.

Infinite loop

In Python, an *infinite loop* is a continuous repetition of a conditional statement until an external factor interferes, such as insufficient CPU memory, failed features or error codes that stop the execution.

When are *infinite* loops necessary?

Infinite loops are not desirable, but they can be useful in certain situations. In client/server programming, an infinite loop may be useful when the server program must run continuously so that client programs can communicate with the server program. You can also use it if you need to create a new connection.

Here is a sample of an *infinite* loop using a *for* loop. The program declares a list and iterates through the list, adding the same element that was in the list before.

```
l = ['XYZ'] #Creating a list

for i in l: #Iterating over the same list
  print(l)
  l.append(i) #Appending the same element
```

The program prints 'XYZ' an endless number of times until it runs out of memory. There is a way to stop the loop by using a *break* statement. In the previous code, we let our program run for as long as it could. To terminate

the execution, we can invoke a *break* statement, say after the length of the list reaches a specific size. In this case, we stop it once the length becomes exactly equal to 20, as illustrated in the code below:

```
#iterating through a loop and adding items in the loop
already
l = ['XYZ'] #Creating a list
for i in l: #Iterating over the same list
  l.append(i) #Appending the same element
  #terminating the loop when a certain condition is met
  if len(l)>20:
    break
print(l)
```

Creating an *infinite* loop

An *infinite* loop can be created using a loop by appending a new element to the list after every iteration.

```
num = [0]
for i in num:
  print(i)
  num.append(i+1)
```

EXAMPLE 9.8

Modify the *infinite* loop above to make sure that it prints a maximum of 20 numbers.

Solution

```
num = [0]
for i in num:
  print(i)
  num.append(i+1)
  if len(num) ==21:
    break
```

9.3.4 Code using a *for* loop with a predetermined number of loops

Sometimes you may want to generate a list of numbers within a range. In this case, a loop becomes more important. For instance, let us assume you want to generate numbers to mimic the Lotto generator. The lotto machine generates six numbers within a range of 1 and 49. You can also use a list that will store the numbers when generated. Every time a new number is added, it will be appended to the list using the *append* method. The code on the next page illustrates how to solve the above problem.

eLINK

~~For more information on~~ loops in Python, ~~visit the following link~~:
futman.pub/Loops

```
#call the random module
import random
mynumbers=[]
#implement a for loop to iterate six times
for x in range(0, 6):
  #add cod to generate random number between 0 and 49
  n = random.randint(1,49)
  #add numbers to the list
  mynumbers.append(n)
#display the lotto numbers in the list
print(mynumbers, end =" ")

Output
[6, 10, 14, 31, 34, 36]
```

You can really create more fun things by sorting the list before it is printed. This can easily be achieved by making use of the `.sort( )` method. Before printing the list, you will add the following snippet:

```
mynumbers.sort( )
```

9.3.5 Using a *for* loop with a non-sequential counter variable as defined by the range function

VOCABULARY

Dictionary – *collection that is ordered, changeable and does not allow duplicates; used to store data values in key-value pairs*

You may need to count the items in a particular data source to determine how frequently they appear. You may, for example, wish to determine how frequently a given item appears in a list or series of values. When your list is small, counting the things is simple and quick. When you have a long list, numbering items becomes more difficult.

A single counter can be used to count the occurrences of a single item. When you need to count numerous different items, you must build as many counters as you have unique objects. A Python **dictionary** may be used to count several things at once.

We are going to use a dictionary to store students' names and their subject marks. Here is the declaration:

```
#dictionary declaration
student_names= {"Andy":34, "Martin":78, "Yolanda":86,
"Memory":56, "Anelisa":64}
#looping through a dictionary
for x in student_names:
  #printing keys and values.
  print(x, student_names[x])
```

The items you want to count will be stored in the dictionary keys. The dictionary values will include the number of times a certain item has been repeated or the object's count.

To count the items in a sequence with a dictionary, for example, you may loop through the sequence, check if the current object is not in the dictionary to initialise the counter (key-value pair), and then increase its count accordingly.

Here is another example that counts the letters in the word “South Africa”:

```
#word to check occurrences
word = “South Africa”
#empty dictionary
counter={}
#implementing a non-sequential counter
for letter in word.upper( ):
  #checking for existence of a letter in the dictionary
  if letter not in counter:
    counter[letter] = 0
  #incrementing the counter
  counter[letter] += 1
print(counter)
```

9.4 *Nested* loops

9.4.1 What is a *nested loop*?

In Python, a loop inside another loop is known as a *nested loop*, i.e. it is a loop inside the body of the outer loop. The *inner* or *outer* loop can be of any type, such as a *while* loop or a *for* loop. For example, the *outer for* loop can contain a *while* loop and vice versa. The *outer* loop can contain more than one *inner* loop. There is no limitation on the chaining of loops. With the help of examples, you will learn how *nested* loops are used in Python.

Syntax of *nested* loops:

```
#outer for loop
for element in sequence
  #inner for loop
  for element in sequence:
    body of inner for loop
  body of outer for loop
```

9.4.2 Defining *inner* and *outer* loops

The *inner* loop will be executed once for each iteration of the *outer* loop. The *outer* loop can contain any number of the *inner* loops. There is no limitation to the nesting of loops.

```
taste = [“red”, “big”, “tasty”]
fruits = [“apple”, “banana”, “cherry”]

for x in taste:
 for y in fruits:
  print(x, y)
```

Output to the code from the previous page:

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

The above example illustrates that the taste "red" is printed several times for each occurrence of fruit.

The *outer* loop can contain more than one *inner* loop. There is no limitation on the chaining of loops. In the *nested* loop, the number of iterations will be equal to the number of iterations in the *outer* loop multiplied by the iterations in the *inner* loop.

In each iteration of the *outer* loop, the *inner* loop executes all its iteration. For each iteration of an *outer* loop, the *inner* loop restarts and completes its execution before the *outer* loop can continue on to its next iteration.

9.4.3 Writing code that will nest identical loops

In this section you will learn about nesting loops whose range is the same. A good example of this is a multiplication table.

EXAMPLE 9.9

Write a nested *for* loop program to print a multiplication table in Python.

```
#outer loop
for i in range(1, 11):
  #nested loop
  #to iterate from 1 to 10
  for j in range(1, 11):
    #print multiplication
    print(i * j, end=' ')
  print("")

Sample output:
1  2  3  4  5  6  7  8  9  10
2  4  6  8  10  12  14  16  18  20
3  6  9  12  15  18  21  24  27  30
4  8  12  16  20  24  28  32  36  40
5  10  15  20  25  30  35  40  45  50
6  12  18  24  30  36  42  48  54  60
7  14  21  28  35  42  49  56  63  70
8  16  24  32  40  48  56  64  72  80
9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90  100
```

9.4.4 Write code that will nest different types of loops

In some instances, the size of an *inner* loop compared to that of an *outer* loop will not be the same. Sometimes the *inner* loop may also rely on the counter for the *outer* loop. See the challenge below.

EXAMPLE 9.10

Write a Python program to construct the following pattern, using a *nested for* loop.

```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

Solution

```
n=5;
for i in range(n):
  for j in range(i):
    print ('* ', end="")
  print('')

for i in range(n,0,-1):
  for j in range(i):
    print('* ', end="")
  print('')
```

9.4.5 Determining the application flow when a *nested* loop statement is encountered

You have learned that we can create multidimensional sequences in which each element is another sequence. How would you iterate through all the values in a multidimensional sequence? You have learned that you must employ loops within loops and when you do this, it is called *nested loops*.

Any Python type that can be iterated over with a *for* loop is an iterable. A list, tuple, string and dictionry are all commonly used iterable types. A tuple or list is iterated over by processing each value in turn.

All these iterables use the same interface for iterating over values, so you can use them almost interchangeably. You can, for instance, iterate over the same list in both levels of a *nested* loop – each loop uses its own iterator, and they do not interfere with each other. As you can see in the code below, we have iterated a list of cars for each element.

```
#list of cars
cars = ['VW', 'ASTRA', 'FORD', 'TOYOTA']
#first loop to pick the first car. This will run an equal
amount of the second loop
for first_car in cars:
  #the second loop is iterating for each of the first car
  selected
  for second_car in cars:
    print("Yesterday I bought a %s. Today I bought a %s." %
    (first_car, second_car))
```

Output

```
My first car I bought is a VW. My second car I bought is at
a VW.
My first car I bought is a VW. My second car I bought is at
a ASTRA.
My first car I bought is a VW. My second car I bought is at
a FORD.
My first car I bought is a ASTRA. My second car I bought is
at a VW.
My first car I bought is a ASTRA. My second car I bought is
at a ASTRA.
My first car I bought is a ASTRA. My second car I bought is
at a FORD.
My first car I bought is a FORD. My second car I bought is
at a VW.
My first car I bought is a FORD. My second car I bought is
at a ASTRA.
My first car I bought is a FORD. My second car I bought is
at a FORD.
```

[why is the wording 'is at a VW/is at a ASTRA, etc.?]

In this program, the use of two iteration variables, namely `first_car` and `second_car`, prints a car for each car in the list. The first loop runs once, and the *inner* loop runs three times.

Activity 9.3 **INDIVIDUAL ACTIVITY**

1. Write a program to find the factorial of a number captured by the user. The *factorial* of a number is the product of all the integers from 1 to that number. For example, the factorial of 5 is 5 × 4 × 3 × 2 × 1 = 120. Factorial is not defined for negative numbers, and the factorial of zero is one, `0!=1`. (10)
2. The *for* loop in Python is used to iterate over a sequence or other iterable objects.
   Which of the following is NOT an example of a *sequence*?
   A List
   B Tuple
   C String
   D None of the above (1)
3. The range function can take three parameters. Give the syntax of range and explain each of the parameters. (7)
4. What will be the output of the following Python code?

```
x = 123
for i in x:
  print(i)
```

   A `1 2 3`
   B `123`
   C `Type error`
   D None of the above (1)
5. What will be the output of the following Python code?

```
for i in range(0):
  print(i)
```

   A `0`
   B `No output`
   C `Error`
   D None of the above (1)
6. Define an *infinite loop*. (2)
7. What will be the output of the following Python code?

```
for i in range(2.0):
  print(i)
```

   A `0.0 1.0`
   B `0 1`
   C `error`
   D None of the above (2)

Activity 9.3 (continued)

8. Consider the code that contains errors given below:

```
1 x = input("Enter value")
2 for k in range[0,20]
3  if x=k
4    print(x+k)
5  else:
6    Print(x-k)
```

The program is supposed to ask the user for a number and print the output as follows:

```
Enter value5
5 4 3 2 1 10 –1 –2 –3 –4
```

Identify the errors in the code and correct them. (8)

TOTAL: 32

Summative assessment

1. Write a program to display the Fibonacci series up to 10 terms.
The *Fibonacci sequence* is a series of numbers. The next number is found by adding the two numbers before it. The *first two numbers are 0 and 1*.
For example, 0, 1, 1, 2, 3, 5, 8, 13, 21. The next number in this series is:
13+21 = 34

Expected output:

```
Fibonacci sequence:
0 1 1 2 3 5 8 13 21 34
```

(14)

2. Write a Python program that accepts a number with two or more digits and prints the reverse of the number.

Example of output:

```
Enter a integer number with more than two digit:
125
User entered number is : 125
The reversed number is: 521
```

(10)

3. What will be the output of the following Python code snippet?

```
x = 2
for i in range(x):
  x -= 2
  print (x)
```

A `0 1 2 3 4 ...`
B `0–2`
C `0`
D `Error` (1)

Summative assessment (continued)

4. What is a *nested loop* in Python? (2)

5. True or false: A *while* loop in Python is used for an indefinite type of iteration. (1)

6. Will the `print( )` statement on line 5 in the code below be executed in this case?

```
a = ['Pretoria', 'Durban', 'Kimberley',
'Bloemfontein', 'Polokwane']
while a:
  print(a.pop( ))
else:
  print('Done.')
```

(1)

7. What is the result of the execution of the following code?

```
for x in range (2):
  for y in range (2):
    print(x, y, sep="")
```

(1)

TOTAL: 30

Module 10

Modularisation and functions

After you have completed this module, you should be able to:

- discuss the concepts *modularisation* and *functions*;
- list the two parts that most Python functions will consist of;
- identify or correct the general form for non-parameter Python functions; and
- write Python code that will:
  - create a non-parameter function that does not return a value
  - call a non-parameter function that does not return a value
  - use one or more non-parameter functions to modify a global variable.

Introduction

Modularity is the process of assembling multiple modules before linking and combining them to form a complete system. Using Python modularisation, we can manage a project by creating modules so that we or other developers can easily understand the code. In this module, you will learn more about modularisation and the non-parameter functions used in Python.

10.1 Modularisation and functions

As you learned in Module 3, variables are used in the development of programs. To recap: Variables are labels for locations in memory used to store values. In statically typed languages, variables have predetermined types, and a variable can only be used to hold values of that type. Python allows us to reuse variables for storing values of any type. When a variable is assigned to an instance, it is mapped to that instance, as variables are references to objects in memory.

To define a new variable in Python, we simply assign a value to a label. The example below illustrates how we create a variable called *count*, which contains an integer value of zero:

```
count = 0
```

If we try to access the value of a variable that has not been defined anywhere yet, the interpreter will exit with a name error. We can define several variables in one line, as shown in the following code snippet, but this is usually considered poor style.

```
#Define three variables at once:
result, number1, number2 = 0, 5, 9

#This is equivalent to:
result = 0
result = 5
total = 9
```

10.1.1 Variable scope and lifetime

The **scope** of a variable in Python is the coding region where the code is visible. In other words, it is the context in which that variable is visible/accessible to the Python interpreter. To access any variable in the code, the scope must be defined. It cannot be accessed from anywhere in the program.

Not all variables are accessible from all parts of the program, and not all variables exist for the same amount of time. Where a variable is accessible and for how long it exists depend on how it is defined. We call the part of a program where a variable is accessible its *scope*, and the duration for which the variable exists its *lifetime*. With regard to scope, we will discuss *global scope*, *local scope* and *built-in scope*.

VOCABULARY

Scope – *context in which a particular variable is visible/accessible to the Python interpreter*

Global scope

A variable that is defined in the main body of a file is called a *global variable*. It will be visible throughout the file and inside any file that imports that file. As a result of their wide-ranging effects, global variables may have unintended consequences. Their use should be minimised. Only objects that are intended to be used globally, such as functions and classes, should be put in the global namespace. If a variable is assigned a value anywhere within the body of the function, it is assumed to be a local unless explicitly declared as global.

Local scope

A variable that is defined inside a function is *local* to that function. It is accessible from the point at which it is defined until the end of the function and exists for as long as the function is executed.

Let us look at the code snippet below:

```
num1=5 #global variable
num2=6 #
def add( ):
    result=num1 + num2
    new_variable=7
    print(result)
print(num1)
print(num2)
add( )
print(new_variable)
```

If we run the code snippet on variables, we will get the following output:

```
  print(new_variable)
NameError: name 'new_variable' is not defined
5
6
11
```

The code managed to get values for `num1` and `num2` and the result of the add function. The input `new_variable` generated an error, because we tried to access it outside the block in which it had been declared. In this case, the scope of `new_variariable` is limited within the `def( )` block. (This will become clear when we discuss the concept of function in section 10.2.)

Built-in scope

This is the broadest scope. The built-in scope contains all the names that are loaded into a Python variable's scope when we start the interpreter. For example, we never need to import any module to access functions such as `print( )`.

Using global keywords in Python

Until now, it has not been necessary to change the global scope in any of the coding examples or exercises. Let us look at an example of where the scope needs to change.

```
num1=1
def counter( ):
    num1=2
    print(num1)
counter( )
```

If we run the code above, the output is 2. However, when we make a reference to 'num1' outside this function, the output is 1 instead of 2. So, what must you do if you want to change the global version of 'num1'? This is when you need to use the *global* keyword in Python.

The modified code will look as follows:

```
num1=1
def counter( ):
     num1=2
    print(num1)
    counter( )
print(num1)
```

Running the code above will give the following result:

```
2
2
```

In the code above, we declared that the 'num1' we are going to use in this function is from the global scope. Once this is done, whenever we refer to 'num1' inside `counter( )`, the interpreter knows we are referring to the global 'num1'.

EXAMPLE 10.1

Using a *nonlocal* keyword in Python:

```
def ages( ):
  my_age=35
  def ourAges( ):
      my_age=55
      ages=55
      print(my_age)
      print(ages)
  ourAges( )
  print(my_age)
ages( )

Output
55
55
35
```

EXAMPLE 10.1 (CONTINUED)

As you can see, this did not change the value of `my_age( )` outside function `ourAges( )`. To accomplish that, we use a *nonlocal* keyword.

```
def ages( ):
  my_age=35
  def ourAges( ):
      nonlocal my_age
      ages=55
      print(my_age)
      print(ages)
  ourAges( )
  print(my_age)
ages( )

Output
35
55
35
```

Note how we managed to use the value `my_age( )` in the `ourAges( )` block, but the value is the same when we print it, i.e. 35.

Activity 10.1 — INDIVIDUAL ACTIVITY

1. Describe the scope of the variables `num1`, `num2`, `num3` and *result* in the following example:

```
def my_function(num1):
  num2 = num1 - 2
  return num2

num3 = 3

if c > 2:
  result = my_function(5)
  print(result)
```

(4)

2. Define the term *scope*. (2)
3. Differentiate between *global scope* and *local scope* with regard to variable declaration. (4)

TOTAL: 10

10.2 Simple Python functions

VOCABULARY

Function – *block of code that only runs when it is called*

A **function** is a reusable block of programming statements designed to perform a certain task. A function block of code runs only when it is called.

Functions in Python provide organised, reusable and modular code to perform a set of specific actions. Functions simplify the coding process, prevent redundant logins and make the code easier to follow. Python has many built-in functions, such as `print( )`, `input( )` and `len( )`. Besides built-ins, you can also create your own functions to do more specific jobs – these are called *user-defined functions*. You can pass data, known as *parameters*, into a function. A function can return data as a result.

Most programming languages today support user-defined functions, although they are not always called *functions*. You may see them referred to as one of the following in other languages:

- Subroutines
- Procedures
- Methods
- Subprograms.

There are several very good reasons for defining functions. Reusability is a fundamental reason why functions should be defined. Modularity is another important aspect of function. By using functions, complex processes can be broken down into smaller steps.

10.2.1 Python functions declaration

To define a function, Python provides the *def* keyword followed by the function name. After the function name comes brackets `( )`. The function may take arguments as input within the opening and closing parentheses, just after the function name, followed by a colon. After defining the function name and argument(s), a block of program statements starts at the next line. These statements must be indented.

Syntax of defining a function:

```
def function_name(parameters):
  statement1
  statement2
  ...
  ...
  return [expr]
```

Let us break down the above declaration in detail:

COMPONENT	MEANING
`def`	The keyword that informs Python that a function is being defined
`function_name`	A valid Python identifier that names the function
`parameters`	An optional, comma-separated list of parameters that may be passed into the function
`:`	Punctuation that denotes the end of the Python function header (the name and parameter list)
`Statement(s)`	A block of valid Python statements

The two new terms mentioned in the function declaration are **arguments** and **parameters**. A *parameter* is the variable listed inside the parentheses in the function definition. An *argument* is the value that is sent to the function when it is called.

Positional arguments are arguments that can be called according to their position in the function call, i.e. values are assigned to the arguments by their position when the function is called. For example, the first positional argument must be first when the function is called. The second positional argument needs to be second, etc.

VOCABULARY

Argument – *value that is passed to a function when it is called*
Parameter – *variable listed inside the parentheses in the function definition*

EXAMPLE 10.2

Take the following as an example: Declare a function to add two numbers.

The function will take in two parameters and return the answer when called.

```
#declaring function with two parameters
def adding_numbers(number1, number2):
    #processing
    answer = number1 + number2
    return answer
#calling the function with two arguments
result = adding_numbers(23, 21)
#displaying output
print(result)
```

Let us analyse the code:

- We defined a function using the keyword *def*.
- The function name is *adding* and inside the brackets we have two parameters – `number1` and `number2` – because we want to add just two numbers. If you try to call the function with 1 or 3 arguments, you will get an error.
- In the body of the function, which follows the colon, we add the statements to be executed.
- Lastly, we use the *return* keyword, which gives back the output when called.

VOCABULARY

Return statement – *special statement that can be used inside a function to end the execution of the function call or return a value from a function*

The *return* statement

What happens when a Python function includes a ***return* statement**? After all, in many cases, if a function does not cause some change in the calling environment, there is not much point in calling it at all. You will notice that unlike in many other languages, you do not need to explicitly declare the return type of the function in Python. Python functions can return values of any type via the *return* keyword. One function can return any number of different types!

How should a function affect its caller? One possibility is to use function return values. A *return* statement in a Python function serves two purposes:

- It immediately terminates the function and passes execution control back to the caller.
- It provides a mechanism by which the function can pass data back to the caller.

In some programming texts, the parameters given in the function definition are referred to as *formal parameters*, and the arguments in the function call are referred to as *actual parameters*.

Calling a function

Calling a function in Python is similar to other programming languages: using the function name, parenthesis (opening and closing) and parameter(s). See the syntax, followed by an example.

In the earlier example we used a function called `adding_numbers`. The function was called as follows:

```
Adding_numbers(23, 21)
```

VOCABULARY

Calling a function – *giving the name of the function and then, followed in parentheses, the argument values (if any) that are needed* `function_name(arg1, arg2)`

We passed two values (23 and 21) inside the parenthesis of a function call. This is because we had two parameters when we declared our function.

It is possible to have a function without a *return* statement. You can have a print with the function body itself, in which case the code for the example above would change as follows:

```
#declaring function with two parameters
def adding_numbers(number1, number2):
    #processing
    answer = number1 + number2
    print(answer)
#calling the function wih two arguments
adding_numbers(55, 36)
#displaying output
```

Working with numbers, it is also possible to pass a sequence into the function as an argument. In the code snippet below, we pass a list of numbers into the function called *sequence*.

```
def sequence(number):
  for i in number:
    print(i, end= " ")
sequence([1,2,3,4,5,6,7,8,9])
```

eLINKS

To learn more about Python functions, visit the following links:

- futman.pub/PythonFunctions
- futman.pub/Pythonforbeginners

Practical activity 10.1 **INDIVIDUAL ACTIVITY**

Modify the sequence function and add all the numbers. The program must print all the numbers as well as the sum.

The output should be as follows:

```
1 2 3 4 5 6 7 8 9
the sum of [1, 2, 3, 4, 5, 6, 7, 8, 9] is 45
```

(12)

TOTAL: 12

Default arguments

VOCABULARY

Default argument – *argument that takes default values if no specific value is passed to it from the function call*

A **default argument** is an argument that takes a default value if an explicit value is not passed to the argument in the function call. Refer to the example below:

```
#declaring function with default parameters
def greetings(name="Zanelle"):
    print("Hello", name)
#calling the function
greetings( )
```

In the above example, the function call has no argument, as a default parameter was assigned in the function heading. However, if we call the function and add a different argument, the output will be with the new name. For instance:

```
#declaring function with default parameters
def greetings(name="Zanelle"):
    print("Hello ", name)
#calling the function
greetings("Tendai")

Output
Hello Tendai
```

Unknown number of arguments

A function in Python can have an unknown number of arguments if you place an asterisk (*) before the parameter when you do not know the number of arguments the user is going to pass.

```
def greeting(*names):
    print ('Hello ', names[0], ', ', names[1])
    return
greeting('Siyanda', 'Martin', 'Tendai')

Output
Hello Siyanda , Martin
```

In this example, you will notice that only the first and second names are printed. The arguments have been affected through the index of the parameters given.

Variable length arguments

Up until now, functions had a fixed number of arguments. In Python, there are other ways to define a function that can take a variable number of arguments. We can pass a variable number of arguments to a function using special symbols.

There are two special symbols:

- `*args` (non-keyword arguments)
- `**kwargs` (keyword arguments).

NOTE

*We use the wildcard or * notation as a function's argument as follows *args OR **kwargs when we have doubts about the number of arguments we should pass into a function." However, we are not going to cover **kwargs, as this is beyond scope of this course.*

EXAMPLE 10.3

Here is an example using a variable length argument:

```
def addition(x, *y):
    print(x)
    for a in y:
   print(a, end=" ")
  return
addition(4,2,3)
print(addition(3))

Output
4
2 3 6
None
```

Let us try to understand the output. The first *print* (*x*) statement gives 4, as 4 is the first parameter passed to the function. On the second line of output, the first value of *y* is 2 and then it prints another 2. The last parameter for *y* is the one passed through the last function call, which in our case is 6. The final print statement yields *None*. You can try this by placing readable descriptions

(called *commenting*) in each of the print statements or the function call. This also changes the arguments of the function. When we call this function, we can pass as many arguments as we wish. They will be accessible in the function as a tuple.

10.2.2 General form for non-parameter Python function

In section 10.1, you learned how a user-defined function is created. The user-defined function has two parts: the *function definition* and the *function call*. The function definition may or may not have parameters and it may or may not return a value. In this section, we look at an example of a Python function with no parameters and no return.

We have created a program to calculate the factorial of a number before. By now, everyone knows the meaning of *factorial*. We are going to use a function to calculate the factorial of a number.

```
#declaring the function
def factorial_number( ):
    '''
    This program calculates
    factorial of a number
    '''

    factorial = 1
      fact_number = int(input("Enter a number for which to
    calculate factorial: "))
    for i in range(1, fact_number + 1):
      factorial = factorial * i
    print("The factorial is", factorial)

##Function Call
factorial_number( )

Output
Enter a number for which to calculate factorial: 5
The factorial is 120
```

Practical activity 10.2 **INDIVIDUAL ACTIVITY**

Write a Python function to check whether a number falls within a given range. The program must allow the user to enter the lower limit and the upper limit of the range, and check whether the number exists in the range.

Output should be as follows:

```
Enter the number to be checked5
Enter the lower limit of the range1
Enter the upper limit of the range9
5 is in the range
```

(10)

10.2.3 Creating a non-parameter function that does not return a value

When functions do not return a value, they are called *void functions*. These functions return *None*. Whenever a function returns a value, it is called a *fruitful function*. You can return something other than *None* by using a return statement in the functions.

In Python, values such as *None*, *True* and *False* are not strings; they are special values and keywords. Whenever we reach the end of a function without explicitly executing a return statement, Python returns *None*. Consider the following functions without a return value:

```
#declaring a function without a return statement
def noReturn( ):
    pass

result=noReturn( )
print(result)

#Output
None
```

As can be seen from the code snippet of the *noReturn* function, the return function gives *None* as the output. We can put as many statements into the function body as we like, perhaps with a print statement, but it will still return *None* even though it can print the statements. This is reflected in the code below:

```
def noReturn(name='Mike', surname=' Magwaza'):
    print(f'My fullname is {name+ surname}')
result=noReturn( )
print(result)

Output
My fullname is Mike Magwaza
None
```

No matter how long or complex your functions are, if they do not have a return statement, or if they have a return statement without a return value, they will return *None*.

NOTE

The Python interpreter does not display None. To show a return value of None in an interactive session, you need to explicitly use print( ).

10.2.4 Call a non-parameter function that does not return a value

In section 10.2.3 you learned how to declare a function with no return value. This was demonstrated by using the `noReturn( )` function. The example

went as far as calling the function, illustrating how such a function is called. We have already mentioned that, when calling such a function, you are required to declare a variable that will hold the output from the function. Displaying the result will require calling the *print* keyword. The program will get an *implicit* return *statement* that uses *None* as a return value if you call the function instead. To reinforce the concept, let us look at the following example:

EXAMPLE 10.4

```
#declaring a function which takes in one parameter
def add_one(x):
    #adding the value passed to 5
    result=x+5
    #displaying the out. NB=> no return value
    print(result)
add_one(8)

Output
13
```

However, if we modify the program a bit and use a variable called *result* to get the return value, we get *None* as the output, as shown below:

EXAMPLE 10.5

```
1  #declaring a function which takes in one parameter
2  def add_one(x):
       #adding the value passed to 5
       result=x+5
       #displaying the out. NB=> no return value
       print(result)
3  result=add_one(8)
4  print(result)

Output
13
None
```

Line 4 (Example 10.5) gives the *None* output, because the program does not have a return value. If you do not supply an explicit return statement with an explicit return value, Python will supply an implicit return statement using *None* as a return value. In the above example, `add_one( )` adds 5 to *x* and stores the value in *result*, but it does not return the result.

10.2.5 Using one or more non-parameter functions to modify a global variable

As you learned in section 10.1.1, a *global* keyword allows a user to modify a variable outside of the current scope. It is used to create global variables from

non-global scopes, i.e. inside a function. The global keyword is used inside a function only when we want to do assignments or when we want to change a variable. The *global* keyword is not needed for printing and access.

Accessing global variable

We use a global keyword to use a global variable inside a function. There is no need to use global keywords outside of a function. In the example below, we did not use the keyword *global* to access `number1` and `number2`, because they are automatically available inside the function body.

```
#global variable
number1 = 50
number2= 60
#function to perform addition
def addition( ):
    result = number1 +number2
    print(result)
#calling a function
addition ( )

Output
110
```

What happens if we want to modify any of the values of the global variables inside the function? The variable can simply be declared as *global* by using the keyword *global* before using the variable. In the code below, a global variable is modified inside a function. The program takes no parameters, but uses global variables.

EXAMPLE 10.6

```
1 #global variable
2 number1 = 50
3 number2= 60
4 #function to perform addition
5 def addition( ):
      global number1
      #modifying the value of the global
      number1=number1 +15
      print(f'The value of number1 inside the function
      is {number1}')
      #processing
      result = number1 +number2
      return result
6 print(f"The value of number1 before change was:
  {number1}")
7 #calling a function
8 answer=addition( )
9 print(number1, "+", number2, "=", answer)

The value of number1 before change was: 50
The value of number1 inside the function is 65
65 + 60 = 125
```

EXAMPLE 10.6 (CONTINUED)

If line 6 were to be placed *after* line 8, you would see that when you change the value inside the function `addition( )`, the change would also be reflected in the value outside the global variable, as shown below:

```
1  #global variable
2  number1 = 50
3  number2= 60
4  #function to perform addition
5  def addition( ):
       global number1
       #modifying the value of the global
       number1=number1 +15
       print(f'The value of number1 inside the function
       is {number1}')
       #processing
       result = number1 +number2
       return result
6  #calling a function
7  answer=addition( )
8  print(f"The value of number1 before change was:
   {number1}")
9  print(number1, "+", number2, "=", answer)

Output
The value of number1 inside the function is 65
The value of number1 before change was: 65
65 + 60 = 125
```

NOTE

If we change the value of the variable inside the function without using the keyword global*, we will get the following error:*

```
number1=number1 +15
UnboundLocalError: local variable 'number1' referenced before
assignment
```

The best way to share global variables across different modules within the same program is to create a special module (often named *config* or *cfg*). However, this is outside the scope of this module.

Using global keyword in nested functions

To use *global* inside a nested function, we must declare a variable with a global keyword inside the nested function, as illustrated in the code below.

```
#function to perform addition with global variables in a
nested function
def addition( ):
  number1=35
  def new_calculation( ):
      #referencing the global variable in addition( )
      function
      global number1
      number1=80

  print(f'Before making change, the value of number1
  is:{number1}')
  new_calculation( )
  print(f'Making change, the value of number1 is:
  {number1}')
addition( )
print(f'After changes, the new value of number1 is :
{number1}')
```

Output

```
Before making change, the value of number1 is: 35
Making change, the value of number1 is: 35
After changes, the new value of number1 is: 80
```

In the above program, the new value of *number1*, which is 80, is only reflected after making changes because we used the keyword *global* inside the `new_calculation( )` function.

Activity 10.2 **INDIVIDUAL ACTIVITY**

1. Write a program to check whether the year captured by a user through the keyboard is a leap year or not. The program must make use of a function.

The output should be as follows:

```
Please Enter a Year: 2024
2024 a Leap Year
```

(10)

2. **2.1** Define the term *function* as used in Python. (2)

2.2 List the TWO common parts of a Python function. (2)

3. What is the output of the following function call?

```
def fun1(num):
    return num + 25
fun1(5)
print(num)
```

A 25

B 5

C NameError 'variable' not defined

D 0 (1)

Activity 10.2 (continued)

4. What is the output of the following function call?

```
def fun1(name, age=20):
    print(name, age)
fun1('Emma', 25)
```

A Emma 25
B Emma 20
C Error
D Emma 25 20 (1)

5. What is the default return value for a function that does not return any value explicitly?

A None
B int
C public
D null (1)

6. Which of the following keywords marks the beginning of the function block?

A `fun`
B `function`
C `def`
D `define( )` (1)

7. Differentiate between Python *parameters* and Python *arguments*. (4)

TOTAL: 22

Summative assessment

1. Differentiate between *formal parameters* and *actual parameters*. (4)

2. Define the term *positional arguments*. (2)

3. Which of the following is the use of function in Python?

A Functions are reusable pieces of programs.
B Functions do not provide better modularity for your application.
C You cannot also create your own functions.
D All of the above. (1)

4. What will the output of the following code be when it is executed? (2)

```
def print_max(a, b):
  if a > b:
    print(a, 'is maximum')
  elif a == b:
    print(a, 'is equal to', b)
  else:
    print(b, 'is maximum')
print_max(3, 4)
```

Summative assessment (continued)

5. True or False: In Python programming, we use *nonlocal* keywords to create nonlocal variables. (1)
6. Name the TWO special symbols used for passing arguments. (2)
7. Write a Python function called `get_average( )` to calculate the average of numbers passed in by the user. The program must make use of variable length arguments (*). (10)

TOTAL: 22

Module 11

Arrays and lists

After you have completed this module, you should be able to:

- define a *data structure*;
- differentiate between a *list*, an *array* and a *dictionary*;
- define a *list* and an *array*;
- use a list in a program;
- use an array in a program;
- perform basic operations on an array;
- perform basic operations on a list; and
- implement the given range of list methods as part of a solution.

Introduction

Computers process and store data accurately and at an exceptionally high speed. It is also essential that the stored data can be accessed quickly. There are different ways of achieving accuracy and speed, such as the implementation of text files, databases and data structures. In this module, you are going to learn more about data structures specific to Python.

11.1 One-dimensional arrays and basic lists

11.1.1 Define data structure

Data structures are a way of organising and storing data in a computer system so that it can be accessed more efficiently depending on the situation. Programming languages are built around data structures. Compared to other programming languages, Python simplifies the learning of these data structures, as you will discover as you work your way through this module.

VOCABULARY

Data structure – *specialised means of organising, processing, retrieving and storing data*

General data structures

In Python, data structures are implicitly supported, which enables you to store and access data easily. Under built-in data structures, the following are included: list, dictionary, tuple, array and set.

NOTE

Python does not have a built-in concept of the array, but you can import it using the array module or the NumPy library. NumPy stands for Numerical Python and is a Python library used for working with arrays. The distinction will be explained a little later.

In Python, users can also create their own data structures, enabling them to customise their functionality. Stacks, queues, trees, linked lists, and so forth, all of which are available to users in other programming languages, fall under the category of *user-defined structures*. In this module, we are going to focus on *lists*, *arrays* and *dictionaries* in detail.

11.1.2 Differentiating between a *list*, an *array* and a *dictionary*

VOCABULARY

Array – *data structure that can store a fixed-sized collection of elements of the same data type*

We are going to explore the differences between **lists**, **arrays** and **dictionaries** here, while the terms *list* and *array* will be defined in section 11.1.3.

Lists are used to store multiple items in a single variable. They are like arrays in other programming languages. Both arrays and lists use indexes

to differentiate between the values stored in a specific order. Different data types can be used for values in a list, e.g. a single list can contain integers, strings as well as objects. Lists can also be changed after their creation.

An *array* is a data structure consisting of a collection of elements (values or variables), each identified by at least one *array index* or **key** (a field used to store data). The first requirement for using an array in Python is to import a special module (*array* or *NumPy*). Lists and dictionaries do not require additional imports, as they are extensions of Python.

Furthermore, an array must contain the same values for all data types (unless you are using *ndarray*). Arrays can also be used to directly handle arithmetic operations. Mathematics can be performed on other data types, but directly as in the case of an array.

A *dictionary* uses key-value pairs to store data values, unlike other data types that hold only a single value. In the pairs of values, one element is the key and the other expresses its value. Values in a dictionary can be of any data type and can be duplicated, whereas keys cannot be repeated and must be immutable (unchangeable). In Python, a dictionary can be created by placing a sequence of elements, separated by a comma, within curly { } brackets.

VOCABULARY

Key – *a field used to sort data/piece of information required to retrieve some data*

Here is an example of how a dictionary is used to store the names of five family members:

```
#declaring a dictionary
family={1:"Godwin", 2:"Rudo", 9: "Tendai", 3:"Tadiwa",
4:"Tendai"}
#displaying dictionary values
print(family)
#printing specific values
print("Record with key 2 is: " + family[2])
print("Record at key 9 is: " + family [9])

Output
{1: 'Godwin', 2: 'Rudo', 9: 'Tendai', 3: 'Tadiwa', 4:
'Tendai'}
Record with key 2 is: Rudo
Record at key 9 is: Tendai
```

List values can be printed without iterating through them all. By contrast, an array would require a loop. A dictionary would also need to be iterated through a loop.

Arrays and lists can be reversed easily without having to iterate through them, which is not possible with a dictionary.

The code below illustrates how to reverse an array or a list using the `reverse( )` function.

```
#reversing array using reverse method
from array import *
arr= array('i',[1, 2, 3, 6, 4, 5])
arr.reverse( )
print("Reverse array",arr)

#Reversing a list
list=[1,2,3,4,5,6]
answer = list[::-1]
#using the reverse method
list.reverse( )
print("Reverse list",(list))

Output
Reverse array array('i', [5, 4, 6, 3, 2, 1])
Reverse list [6, 5, 4, 3, 2, 1]
```

11.1.3 Defining *lists* and *arrays*

Python lists

A *Python list* is an ordered and changeable collection of data objects (refer to section 11.1.2 above). It is the most versatile data type available in Python. It can be written as a list of comma-separated values (items) between square brackets. A list is defined as follows:

```
List name=[element, element]
```

Here is an example in implementation:

```
list_one=["Ziyanda", "Stephen", 200, 'A', 215.50, 'Uzair']
print(list_one)
#checking the type
print(type(list_one))
#printing specific items
print(list_one[1])
```

Lists are ordered when they are entered, but they can be changed later. Here is an illustration of two lists:

```
list_one=["Ziyanda", "Stephen", 200, 'A', 215.50, 'Uzair']
list_two=["Ziyanda", "Stephen", 200, 'A', 215.50, 'Uzair']
print(list_one==list_two)

Output
True
```

If we swap the position of elements in the list above while maintaining the same size and checking for equality, the answer will be *False*, because the position of value 200 has been changed, as indicated on the next page.

```
list_one=["Ziyanda", "Stephen", 200, 'A', 215.50, 'Uzair']
list_two=["Ziyanda", 200,"Stephen",  'A', 215.50, 'Uzair']
print(list_one==list_two)

Output
False
```

Lists are also indexed, starting at zero, and counting so you can iterate through the list. Because an implicit index is given to each value, you can have duplicate values within the list. A unique advantage of a Python list is that it can store items of different types.

An array

VOCABULARY

Index – *method of sorting data by creating keywords or a listing of the data*

Element – *single part of a larger group*

As mentioned in section 11.1.2, an *array* is a data structure that stores values of the same data type. Most data structures make use of arrays to implement their algorithms. It is important to understand what we mean by the terms **index** and **element**, as these are central in array usage. Each item stored in an array is called an *element*. The location of each element is identified by an *index* and has a numerical value.

Array representation

Take the following situation as example. A lecturer wants to store the marks of 10 students. The marks are: 65, 89, 74, 83, 52, 41, 36, 76, 84, 39. The representation would look as follows:

FIRST INDEX									LAST INDEX
0	1	2	3	4	5	6	7	8	9
65	89	74	83	52	41	36	76	84	39

The highlighted values are the *elements*. The row above the elements indicates the *index*. Each element can be accessed via its index. For example, we can fetch an element at index 6 as 36.

The basic operations supported by an array are the following:

- Traverse – prints all the array elements one by one
- Insertion – adds an element at the given index
- Deletion – deletes an element at the given index
- Search – searches an element using the given index or by the value
- Update – updates an element at the given index.

An array in Python can be created by importing the *array* module: `array(data_type, value_list)`, which is used to create an array with the data type and value list specified in its arguments.

Syntax of array declaration:

```
from array import *

arrayName = array(typecode, [Initializers])
```

The `typecode( )` function defines the type of value an array will hold. The table below shows some common type codes and their values.

TYPECODE	VALUE
'i'	Represents signed integer of size 2 bytes
'I'	Represents unsigned integer of size 2 bytes
'f'	Represents floating point of size 4 bytes
'c'	Represents character of size 1 byte
'u'	Represents unicode character

Table 11.1.1: Types of codes used in arrays

EXAMPLE 11.1

Array implementation

We are going to use Python to store the 10 marks given earlier in this section. Here is an example using the *array module*.

```
from array import *

marks = array('i', [65, 89, 74, 83, 52, 41, 36, 76, 84,
39])
print(marks)
print(marks[4])
print(type(marks))

Output
array('i', [65, 89, 74, 83, 52, 41, 36, 76, 84, 39])
52
<class 'array.array'>
```

If we replace any of the elements with a string or character, the program will generate an error. For this reason, the array module allows us to store elements of the same type, which fits our definition of an array.

EXAMPLE 11.2

We can implement the same concept using the *NumPy library* and the following syntax:

```
import numpy as np
arr = np.array([65, 89, 74, 83, 52, 41, 36, 76, 84, 39])
print(arr)
print(type(arr))

Output
[65 89 74 83 52 41 36 76 84 39]
65
<class 'numpy.ndarray'>
```

The function `ndarray` is an array object in *NumPy*, which is created by using the `array( )` function. Check the output type in Example 11.2. It differs slightly from the output in Example 11.1. When using *NumPy*, if you add

the *typecode*, the code will generate an error. In addition, you can now add a string as an element and the code will still run. This is contrary to our definition of an array that stores elements of the same type.

A *numpy.array* is just a convenient function to create an *ndarray*; it is not a class in itself. We recommend constructing arrays using the first option if the type of elements is very important in the implementation.

Array dimensions

A *one-dimensional array* is the simplest form of an array in which the elements are stored linearly and can be accessed individually by specifying the index value of each element stored in the array. When it comes to dimensions, *NumPy* is easier to implement, because it can use built-in functions such as `linspace( )`, `array( )` and `arrange( )`.

11.1.4 Using a list in a program

The example below shows how to traverse through a list, calculate the sum as well as find the highest and smallest items in the list:

EXAMPLE 11.3

```
marks=[34,2,67,12,98,56,23,65]
for k in range(len(marks)):
  print(k, marks[k])
print(f'The sum of the list is : {sum(marks)}')
print(f'The highest mark in the list is : {max(marks)}')
print(f'The lowest mark in the list is : {min(marks)}')

Output
marks=[34,2,67,12,98,56,23,65]
for k in range(len(marks)):
  print(k, marks[k])
print(f'The sum of the list is : {sum(marks)}')
print(f'The highest mark in the list is : {max(marks)}')
print(f'The lowest mark in the list is : {min(marks)}')
```

11.1.5 Using an array in a program

In the following example, an array will be used to store integer numbers. We are going to declare an array called *my_numbers* and reverse the array. The program will print the original array and the reversed array as well as occurrences of a chosen element.

EXAMPLE 11.4

```
Sample:
[1, 3, 5, 3, 7, 1, 9, 3])
The output should be [3, 9, 1, 7, 3, 5, 3, 1])
```

EXAMPLE 11.4 (CONTINUED)

```
from array import *
#declaring the array with numbers
my_numbers = array('i', [1, 3, 5, 3, 7, 1, 9, 3])
#displaying the original array
print("Original array: "+ str(my_numbers))
#reversing the array
my_numbers.reverse( )
print("Reverse the order of the items:"  + str(my_numbers))
print("Number of occurrences of the number 3 in the said
array: "+str(my_numbers.count(1)))
```

Output

```
Original array: array('i', [1, 3, 5, 3, 7, 1, 9, 3])
Reverse the order of the items: array('i', [3, 9, 1, 7,
3, 5, 3, 1])
Number of occurrences of the number 3 in the said array: 2
```

In the above example, we used the `count` function to count occurrences of an element.

Practical activity 11.1 **INDIVIDUAL ACTIVITY**

Write a program to declare an array of letters and search for a specific element. If the element exists, the program must print 'found'; otherwise, 'not found' must be printed.

Sample output

```
Enter the letter to be searched :  p
array('u', 'zkmjy')
The letter p has not been found in the array
```

(10)

TOTAL: 10

11.1.6 Basic operations on an array

Here you are going to learn a few operations that can be performed on an array. For the purpose of this explanation, we will use an array called *marks* to store the following student marks:

```
[52, 63, 84, 74, 90, 41, 44, 65, 21, 34]
```

Range

We can use a *for* loop with range to iterate elements of an array, as shown below:

```
from array import *
marks=array('i', [52, 63, 84, 74, 90,  41, 44, 65, 21, 34])
for k in range(len(marks)):
  print(k, marks[k])
```

```
Output
0 52
1 63
2 84
3 74
4 90
5 41
6 44
7 65
8 21
9 34
```

The *for* loop with range is quite handy if you also want to print the index of the element.

Traversing

Another term for *traverse* is *loop through*. You can use the `for ... in` loop to loop through all the elements of an array, as shown below:

```
from array import *
marks=array('i', [52, 63, 84, 74, 90,  41, 44, 65, 21, 34])

for k in marks:
  print(k)
```

If we loop through the array, elements are printed one at a time, unlike when we execute `print(marks)`. We can also use the `len( )` method to return the length of an array (the number of elements in an array).

`print(len(marks))` will produce 10, as our array has 10 elements.

Replace a value/elements

To replace elements in an array, use the index and the value to be replaced.

Syntax:

```
from array import *

arrayName[index] =new_element
```

Here is the implementation:

```
from array import *
marks=array('i', [52, 63, 84, 74, 90, 41, 44, 65, 21, 34])
marks[2]=99
print(marks)

Output
array('i', [52, 63, 99, 74, 90, 41, 44, 65, 21, 34])
```

In the code above, we managed to replace the element 63 at index *2* with value 99.

Inserting elements into an array

We can use the `insert( )` function to add an element into an array. The function takes two parameters. The first argument is the index at which

the element is to be inserted and the second argument is the element to be inserted. If we want to insert 55 at index 1 in the array below, the code will be as follows:

```
from array import *
marks=array('i', [52, 63, 84, 74, 90,  41, 44, 65, 21, 34])
marks.insert(1, 55)
print(marks)

Output
array('i', [52, 55, 63, 84, 74, 90, 41, 44, 65, 21, 34])
```

Another alternative would be to use `append( )`. It appends element(s) to the end of the array by using the `append( )` function. The `extend( )` function adds each value from a second list as its own element. So, extending a list with another list combines their values. We can even join two arrays by using `extend( )`, as shown below. If you declare another array with two values, 22 and 77, the length of the array becomes 12.

```
from array import *
marks=array('i', [52, 63, 84, 74, 90,  41, 44, 65, 21, 34])
marks2=array('i', [22, 77])
marks.extend(marks2)
print(f'The new length of the array is {len(marks)}')
print(marks)

Output
The new length of the array is 12
array('i', [52, 63, 84, 74, 90, 41, 44, 65, 21, 34, 22, 77])
```

Sometimes you may want to remove specific elements from the array. Python uses `pop( )`, which takes one argument, and this will be the index of the item to be removed from the array.

EXAMPLE 11.5

```
from array import *
marks=array('i', [52, 63, 84, 74, 90,  41, 44, 65, 21,
34])
marks.pop(2)
print(marks)

Output
array('i', [52, 63, 74, 90, 41, 44, 65, 21, 34])
```

Sum of elements in an array

Calculating the sum of all the elements in the array is a bit easier. You can use a loop and add each value to the sum as the program iterates through the loop. To calculate the average (mean), divide the sum by the total number of elements in the loop. In this example, we implement the `len( )` method.

Example of calculating the sum and average using arrays:

```
from array import *
marks=array('i', [52, 63, 84, 74, 90,  41, 44, 65, 21, 34])
#initialise the value of sum to 0
sum=0
#loop through the array and add each value to sum
for k in marks:
  sum=sum+k
print(f'The sum of the array elements is {sum}')
average=sum/len(marks)
print(f'The average of the array elements is {average}')
```

If we simply want to do statistical calculations on the array, using *NumPy* will be much quicker. We can use functions such as:

`sum( )` to calculate sum of all elements; and

`mean( )` to calculate the mean for the array.

Example

```
import numpy
from array import *
marks=array('i', [52, 63, 84, 74, 90,  41, 44, 65, 21, 34])
#calculating the sum of all elements
total=numpy.sum(marks)
#calculating the average of all elements
average=numpy.mean(marks)
print(f'The sum of all elements is {total}')
print(f'The average of all elements is {average}')

Output
The sum of all elements is 568
The average of all elements is 56.8
```

Calculating mode

Mode refers to the most-repeated element or value in the array. You can import *NumPy* and use the `mode( )` function in the Python *statistics* module. Like the NumPy module, the *statistics* module also contains statistical functions such as mean, median, mode, etc.

The following is an example of a mode using the *statistics* module. The array has been modified array so that it contains repeating values.

We will use the following as our new array for calculating mode:

```
marks= [52, 63, 84, 74, 90, 90, 44, 65, 44, 44]

import numpy
import statistics as st
from array import *
marks=array('i', [52, 63, 84, 74, 90, 90, 44, 65, 44, 44])
```

```
#calculating the mode
x=st.mode(marks)
#displaying the result
print(x, "occurs", marks.count(x) , 'times in the array')

Output
44 occurs 3 times in the array
```

Swapping elements

So far, a variety of operations has been performed on arrays in Python. Before we look at swapping elements, you must understand the term. *Swapping* is a process in which two variables exchange the values that they hold.

We are going to discuss two swapping methods:

- Direct swapping
- Using a third temporary variable.

Direct swapping

If the positions of the elements to be swapped are known, we can simply swap the elements using comma assignment. In the example below, a function called *swap_elements* is used, which takes three parameters.

```
#Swapping elements
from array import *
#Swap function
def swap_elements (array, pos1, pos2):

  array[pos1], array[pos2] = array[pos2], array[pos1]
  return array

#The initial array
marks=array('i', [52, 63, 84, 74, 90,  41, 44, 65, 21, 34])
print(marks)
pos1, pos2 = 1, 3
#calling the array with swapped positions
print(swap_elements(marks, pos1-1, pos2-1))

Output
array('i', [52, 63, 84, 74, 90, 41, 44, 65, 21, 34])
array('i', [84, 63, 52, 74, 90, 41, 44, 65, 21, 34])
```

The program above shows an array with 10 elements. Value 52 at index 0 and value 84 at index 2 have been swapped. Remember, positions were used. To turn these into arguments, you must subtract 1 for both *pos1* and *pos2*.

Swapping using a third temporary variable

In this approach, a third variable is used to swap the values present at the given position.

The logic behind swapping two numbers using a third variable is as follows:

- Store the value of the first variable in the third variable, which is a temporary variable.

- Store the value of the second variable in the first variable.
- Lastly, store the value of the third variable in the second variable.

The code for swapping is as follows:

```
from array import *
#Swap function
def swap_array(ourmarks,pos1,pos2):

  #Swapping using third variable temp
  temp = ourmarks[pos1]
  ourmarks[pos1] = ourmarks[pos2]
  ourmarks[pos2] = temp
  return ourmarks
marks=array('i', [52, 63, 84, 74, 90,  41, 44, 65, 21, 34])
pos1= 2
pos2= 5
print (f'The original array is {marks}')
print("Swapped array: ",swap_array(marks,pos1-1,pos2-1))

Output
The original array is array('i', [52, 63, 84, 74, 90, 41,
44, 65, 21, 34])
Swapped list: array('i', [52, 90, 84, 74, 63, 41, 44, 65,
21, 34])
```

Searching

Python has a method to search for an element in an array, known as `index( )`. The position where the element is located in the array can be determined by using the *index* function.

```
from array import *
marks=array('i', [52, 63, 84, 74, 90, 41, 44, 65, 21, 34])
print(marks)
print(marks.index(74))
```

If the value to be searched is not in the array, you will get a *Valueerror x*, not an array statement. There are search algorithms such as binary search, linear search, interpolation search and many others. However, the details of these search techniques lie outside the scope of this curriculum.

Slicing arrays

Python supports the slicing of arrays. *Slicing* is the creation of a new sub-array from the given array based on the user-defined starting and ending indices. Different ways to slice arrays are provided here.

Syntax used:

```
arr[ start : stop : step ]
```

- `start` is the starting index from which we need to slice the array (*arr*).
- `stop` is the ending index, before which the slicing operation will end.
- `step` is the steps the slicing process will take from *start* to *stop*.

Here is an example:

```
from array import *
marks=array('i', [52, 63, 84, 74, 90,  41, 44, 65, 21, 34])
s=marks[2:5]
print(s)

Output
array('i', [84, 74, 90])
```

As can be seen in the code above, only three elements have been extracted from the array starting at *index 2* and going up to *index 5*. So, the *stop* index will always print an element that is one index less. In the case of the given example, the element 90 with index 4 was the last to be printed, as our code has 5 as the stop index. If we use one parameter as the *stop* index, the output will be from index 0 stepping 1.

The code snippet would look as follows:

```
s=marks[:5]

Output
array('i', [52, 63, 84, 74, 90])
```

If we use the *start* index only, the code will print all elements starting from the given index to the end of the array.

Using slicing with a step parameter

When all three parameters are mentioned, you can perform array slicing in Python from index *start* to *(stop-1)* with each index jump equal to the given *step*.

Here is an example of the marks array with a *step* value of 2:

```
from array import *
marks=array('i', [52, 63, 84, 74, 90,  41, 44, 65, 21, 34])
s=marks[
::2]
print(s)

Output
array('i', [52, 84, 90, 44, 21])
```

The code above starts from index 0 to index 9, leaving one element in-between.

11.1.7 Basic operations on a list

So far, we have looked at array implementation together with the common operations that can be performed on arrays. Lists are basically equal to arrays in other programming languages. However, they do have the extra benefit of being dynamic in size. The operations that can be performed on lists are now discussed.

Traversing

There are several methods that can be used to traverse through a list. These include the `range( )` method, *for* loop, *while* loop and many others.

Example of using a *for* loop:

```
#Python3 code to iterate over a list
my_list = [2, 4, 6, 8, 10]
sum=0
#Using for loop
for i in my_list:
  sum=sum +i
  print(i, end=' ')
average=sum/len(my_list)
print(f'\nThe sum of the list is {sum}')
print(f'The average of the list is {average}')
#counting the values in the list
print(f'The length of the list is {len(my_list)}')

Output
2 4 6 8 10
The sum of the list is 30
The average of the list is 6.0
The length of the list is 5
```

Replace a value/elements

Replacing an element requires the use of the index of the element to be replaced and the item to be replaced, as shown in the example below:

```
my_list = [2, 4, 6, 8, 10]
my_list[1]=5
print(f'\The new list is {(my_list)}')
```

Counting items in a list

As mentioned before, we can count the number of items by using `len( )`. We can also use *counts* to check the occurrence of a specific item in the list, as shown in the code below:

```
#list declaration
mynumbers = [5, 1, 2, 2, 4, 3, 1, 2, 3, 1, 1, 5, 2]
counts=mynumbers.count(5)
print(counts)

Output
2
```

As you can see, the code above only shows the occurrence of one item. There are several ways to count the occurrence of each item in the list. In this module, we will look only at two:

- Looping through the list
- Using *collections.counter*.

Looping through the list

This method is similar to the one above, except that `count` is used in a loop and the result is stored in a dictionary every time it counts an item. See the implementation on the next page.

```
#declare an empty dictionary
mydict={}
sum=0
#Add a for loop
for k in mynumbers:
  #count occurence of item k
  counts=mynumbers.count(k)
  #adding each item to sum
  sum=sum+k
  #add key and values to the dictionary
  mydict[k]=counts
#print the keys and values
print(mydict)
#printing the sum of items
print(f'The sum of items in the list is {sum}')

Output
{5: 2, 1: 4, 2: 4, 4: 1, 3: 2}
The sum of items in the list is 32
```

As you can see, there are two occurrences of item 5, four of item 1, four of item 2, one of item 4 and two of item 3. As for the sum, we simply initialised a variable sum, which then stores the incremental sum of *k* values in the loop.

In Python we can find the average of a list by simply using the `sum( )` and `len( )` functions. If we want to be a bit fancy, we can use a function `return sum( )` method, then divide by the length of the list. This implementation is shown below:

```
#declation a function to find average
def Average(mylist):
  return sum(mylist) / len(mylist)
#the list declaration
mynumbers = [5, 1, 2, 2, 4, 3, 1, 2, 3, 1, 1, 5, 2]
#calling the function
average = Average(mynumbers)

#Printing average of the list rounding the result to 2
decimal places
print("Average of the list =", round(average, 2))

Output
Average of the list = 2.46
```

Using collections counter method

We will need to import the module *collections* and use the `counter( )` method to count each item. The result is still shown as a dictionary, just like in the previous example.

```
#import the collections
import collections
#list declaration
mynumbers = [5, 1, 2, 2, 4, 3, 1, 2, 3, 1, 1, 5, 2]
```

```
#check the number of items in the list
print(f'The number of items in the list is
{len(mynumbers)}')
#counting occurence of each item
counter=collections.Counter(mynumbers)
#result is printed in form of a dictionary with keys being
the numbers
#and values being the occurences
print(counter)

Output
{5: 2, 1: 4, 2: 4, 4: 1, 3: 2}
```

Calculating mode in a list

The `mode( )` function in the Python *statistics* module takes some dataset as a parameter and returns its *mode* value, as shown below.

```
#Python3 code to calculate mode
from statistics import mode
my_list = [2, 4, 6,6,6, 8, 10]
print(f'\nThe mode of the list is {mode(my_list)}')
```

[MISSING: FIND/CALCULATE AVERAGE]

Swapping elements

To swap elements in a list, use the same procedure as for arrays. You can also use another method such as `pop( )` method and `insert( )` in a function. Refer to the following examples of programs used to swap items in a list:

```
#creating a function
def swap_pos(mylist, a, b):
#popping the elements from list
  first_element=mylist.pop(a)
  second_element=mylist.pop(b-1)
  #inserting in the respective positions
  mylist.insert(a, second_element)
  mylist.insert(b, first_element)
  return mylist
#initialising the list
my_list = [2, 4, 6, 8, 10]
#displaying the original list
print("The values in the original list are: ", my_list)
#specifying the positions
a, b=2, 5
#passing the values in function
print("The list with swapped elements is: ", swap_pos(my_
list,a-1,b-1))

Output
The values in the original list are: [2, 4, 6, 8, 10]
The list with swapped elements is: [2, 10, 6, 8, 4]
```

Searching

To find an element in a list, use the Python `list.index( )` method. The `index( )` function is an inbuilt Python method that searches for an item in a list and returns its index or position. If the same element is present more than once, the method returns the index of the first occurrence of the element.

```
my_list = [2, 4, 6, 8, 10]
my_element=my_list.index(6)
print(my_element)
```

11.1.8 Implementing list methods as part of a solution

Clear

There are times when you may want to clear the contents of a list. In such cases you will have to use the `clear( )` method.

Syntax:

```
List.clear( )
```

Copying a list

The `copy( )` method returns a shallow copy of the list.

Syntax:

```
newlist=list.copy
soccer_stars=['Maradona', 'Messi', 'Pele', 'Mlla',
'McCarthy']
new_stars=soccer_stars.copy( )
print(new_stars)
```

Python remove

The `remove( )` method is used to remove the first matching element (which is passed as an argument) from the list.

Sorting lists

To sort the list, Python employs the `sort( )` method.

Syntax:

```
List.sort( )
```

EXAMPLE 11.6

Write a program to declare the following lists:

```
numbers = [1, 2, 3, 4, 5, 6, 7]
my_newnumbers=[9,8]
```

- The program must print the squares of the first list.
- Insert a value 10 in the second array in position 1 and print the sorted lists.
- Combine the two lists using *my_newnumbers* as the resulting list. Sort the array and print the result.
- Clear the contents of numbers [ ].

EXAMPLE 11.6 (CONTINUED)

```
numbers = [1, 2, 3, 4, 5, 6, 7]
my_newnumbers=[9,8]
my_newnumbers.insert(1, 10)
#result list
res = []
for i in numbers:
  #calculate square and add to the result list
  res.append(i * i)
print(f'The square of first list are {res}')

print(f'The second array is {my_newnumbers}')
my_newnumbers.extend(numbers)
my_newnumbers.sort( )
print(my_newnumbers)

my_newnumbers.reverse( )
print(f'The reversed list is {my_newnumbers}')
numbers.clear( )
print(f'The contents of numbers are {numbers}')
The square of first array are [1, 4, 9, 16, 25, 36, 49]
The second array is [9, 10, 8]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
The reversed list is [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
The contents of numbers are []
```

Practical activity 11.2 — INDIVIDUAL ACTIVITY

Write a program to count positive and negative numbers in a list. Use the following list:

```
my_numbers = [54, 11, -9, -12, -24, 77, 4, -45, 66,
-39, 1]
```

(10)

TOTAL: 10

Activity 11.1 — INDIVIDUAL ACTIVITY

1. Define the term *data structure* as applied in programming. (2)
2. Define the following terms as applied in Python programming:
   - **2.1** Array
   - **2.2** List
   - **2.3** Dictionary (3 × 2) (6)
3. Differentiate between an *array* and a *list* in Python. (4)
4. List any FIVE basic operations that can be carried out on an array. (5)
5. Using an example, explain how *arrays* are declared in Python. (4)

Activity 11.1 (continued)

6. Write a Python program to multiply all the items in a list. Use a function to solve the challenge. Use the following list of numbers:

```
numbers=[2,3,4,5,6]

Sample output
The total of the list is 720
```

(9)

TOTAL: 30

Summative assessment

1. A list can contain … elements.
   - **A** 100
   - **B** 1000
   - **C** 1000 000
   - **D** Unlimited (depends on the computer's memory) (1)
2. The `list.pop( )` function will …
   - **A** remove the first element from the list.
   - **B** remove the last element from the list.
   - **C** not remove an element from the list.
   - **D** remove both elements from the list. (1)
3. If ['one', 'four', 'three', 'two'] is a *list* object, what will be returned by the `sort( )` function? (3)
4. Explain how you can check the number of items in a list. (4)
5. What is the difference between `append( )` and `extend( )` as applied in Python lists? (4)
6. Given a declaration of a list containing the fruit listed below. Explain how you can find the index of the first matching element, for instance 'apple'.

```
fruits = ['pear', 'orange', 'apple',
'grapefruit', 'apple', 'pear']
```

(2)

7. Consider the following declaration of fruits:

```
fruits = ['pear', 'orange', 'apple',
'grapefruit', 'apple', 'pear']
```

Write a program to display the *element* and the *index* of each fruit. (5)

8. Write a one-line code snippet to generate a list of every integer between 5 and 50 values with a step value of 3.

   The output must look as follows:

```
[5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38,
41, 44, 47]
```

(2)

9. Write a Python program to reverse the order of the items in the following array:

```
Sample Output
Original array: array('i', [1, 3, 5, 3, 7, 1, 9, 3])
Reverse the order of the items:
array('i', [3, 9, 1, 7, 3, 5, 3, 1])
```

(10)

TOTAL: 32

Glossary

A

Algorithm – procedure used for solving a problem or performing a calculation; an algorithm acts as an exact list of instructions

Argument – value that is passed to a function when it is called

Array – data structure that can store a fixed-sized collection of elements of the same data type

Assembler – program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform basic operations

B

Base – a number base is the number of digits or combination of digits that a system of counting uses to represent numbers

Block – piece of Python program text that is executed as a unit

Block-based programming – uses a drag-ang-drop learning environment where programmers use coding instruction blocks to construct animated stories and games

Boolean – system of algebraic notation used to represent logical propositions by means of the binary digits 0 (falses) and 1 (true)

Branch – instruction that can cause a computer to begin executing in a different instruction sequence

Bug – coding error in a computer program

Bytecode – low-level, platform-independent representation of the source (Python) code

C

Calculational thinking – interrelated set of problem solving methods that involve expressing problems and their solutions in ways a computer can also execute

Calling a function – giving the name of the function and then, followed in parentheses, the argument values (if any) that are needed

Character – single visual object used to represent, numbers or symbols

Comment – line in the code that is ignored by the interpreter during the execution of the program

Compiler – special program that translates a programming language's source code into a set of machine-language instructions that can be understood by the computer's processor

Computer – electronic device that accepts data as an input and processes it according to instructions to produce an output; computers are made up of hardware and software

D

Data structure – specialised means of organising, processing, retrieving and storing data

Decimal – data type that uses the base-10 numbering system

Decision table – concise visual representation for specifying which actions to perform depending on given conditions

Default argument – argument that takes default values if no specific value is passed to it from the function call

Deployment – all the processes involved in getting new software (or hardware) up and running properly

Dictionary – collection that is ordered, changeable and does not allow duplicates; used to store data values in key-value pairs

E

Element – single part of a larger group

Exponentiation – operation where a number is multiplied several times by itself

Expression – string that will be evaluated as Python code

F

Feature – unit of software that provides a specific function or capability for the software

Floor division – operation that divides two numbers and rounds the result down to the nearest integer

Flow chart – diagram that shows the logic of the program or sequence of instructions in a single program

Function – block of code that only runs when it is called

G

Global – dictionary containing global parameters with scope throughout the program

I

Identity (of object) – unique, constant integer that exists for the length of the object's life

Import – to bring a file from a different program into the one you are using

Indentation – spaces at the beginning of a code line

Index – method of sorting data by creating keywords or a listing of the data

Input – what the user needs to punch into the computer through the keyboard for the program to start processing

Instructions – accepted by the computer as input; a series of instructions is processed one by one

Integer – data type used to represent real numbers that do not have fractional values

Integrated development environment – software for building applications that combine common developer tools into a single graphical user interface

Interactive mode – back-and-forth engagement between a computer and the user

Interpreter – computer program that directly executes instructions written in a programming language (does not require the program to be converted into machine language)

Iterable – object that can be looped over or iterated over with the help of a *for* loop

Iteration – process where the design of a product or application is improved by repeated reviewing and testing

K

Key – field used to sort data/piece of information required to retrieve some data

L

Lexis – collection of words, phrases and idioms used in a programming language

List – number of items in an ordered or unordered structure

Local – dictionary containing local parameters as a mapped object

M

Modular design – design principle that divides a system into smaller parts known as modules that can be created, modified, replaced or exchanged independently with other modules or between different systems

Module – any of several distinct but interrelated units from which a program may be built up/into which a complex activity may be analysed

Modulus – also referred to as modulo; the remainder from the division of the first argument to the second

N

Newline – character used to represent the end of a line of text and the beginning of a new line; also known as a line break or end-of-line marker

P

Parameter – variable listed inside the parentheses in the function definition

Parse – to analyse an object specifically/read the program code

Problem solving – defining the problem, determining its root cause, identifying, prioritising, and deciding on options, as well as implementing the solution

Pseudocode – detailed, readable description of what a computer program or algorithm must do, expressed in a natural language rather than a programming language

R

Radix – number of unique digits, including the digit zero, used to represent numbers

Recursive – method of solving a calculational problem where the solution depends on solutions to smaller instances of the same problem

Relational operators – symbols used to compare the operands on either side and determine the relationship between them

Repository – database of changes

Return statement – special statement that can be used inside a function to end the execution of the function call or return a value from a function

Revision control – class of systems responsible for managing changes to computer programs, documents, large websites or other collections of information

Rollback – process of returning a software program to an earlier version

Runtime – final stage of the program lifecycle, when the machine executes the program code

S

Scope – context in which a particular variable is visible/accessible to the Python interpreter

Script – program or sequence of instructions that are interpreted or carried out by another program, rather than by the computer processor

Script mode – used when the user is working with more than one single code or a block of code

Selection – programming construct where a section of code is run only if a condition is met

Sequence – default control structure; instructions are executed one after another

Shell – program that presents a command line interface that enables you to interact with your computer by entering commands via a keyboard; exposes the services of an operating system to a human user or other programs

Slicing – feature that enables accessing parts of sequences such as strings, tuples and lists; can also be used to modify or delete the items of changeable sequences such as lists

Software – set of instructions, data or programs used to operate computers and execute specific tasks

Software author – person who has authored or participated in the development of the software or any portion thereof

Sprint – set amount of time that a development team has to complete a specific amount of work

Sprite – two-dimensional image or animated image that is integrated into a larger environment and often plays a specific role

Statement – instruction that the Python interpreter can execute

String – data values that are made up of ordered sequences of characters, such as “Hello World”

Syntax – rules that define the combinations of symbols that a computer’s processor can interpret

T

Timestamp – sequence of characters or encoded information identifying when a certain event occurred

Token – basic component of source code

Top-down programming – programming style in which the design begins by specifying complex pieces and then dividing them into successively smaller pieces

Trunk – unnamed version of a file or program that is being processed under revision control

V

Validation – checking or verifying data against a given set of rules before it enters the computer system

Variable – value that can vary, in other words, change over time and under the control of the program

Version-control system – system that tracks changes to a file or set of files over time

Visual programming – programming language that uses graphical elements and figures to develop a program

References

Polya, G. (1957). *How To Solve It, by George Polya*.
https://www.productplan.com/glossary/user-story/

Marji, M. (2014). *Learn to program with Scratch*.
https://nostarch.com/download/samples/Learn-Scratch-05.pdf

Python Notes for Professionals: GoalKicker.com

Dawson, M. (2010). *Python programming for the absolute beginner* (p. 480).
Boston, MA: Course Technology.

Yates, J. (2016). *Python: Practical Python Programming For Beginners and Experts*
(Beginner Guide).

Loads of repeats
here
and must be edited

Python Notes for Professionals: GoalKicker.com

Dawson, M. (2010). *Python programming for the absolute beginner* (p. 480).
Boston, MA: Course Technology.

Marji, M. (2014). *Learn to program with Scratch*.

Yates, J. (2016). *Python: Practical Python Programming For Beginners and Experts*
(Beginner Guide).

Python Notes for Professionals: GoalKicker.com

Dawson, M. (2010). *Python programming for the absolute beginner* (p. 480).
Boston, MA: Course Technology.

Yates, J. (2016). *Python: Practical Python Programming For Beginners and Experts*
(Beginner Guide).

Python Notes for Professionals: GoalKicker.com

Dawson, M. (2010). *Python programming for the absolute beginner* (p. 480).
Boston, MA: Course Technology.

Yates, J. (2016). *Python: Practical Python Programming For Beginners and Experts*
(Beginner Guide).

Python Notes for Professionals: GoalKicker.com

Dawson, M. (2010). *Python programming for the absolute beginner* (p. 480).
Boston, MA: Course Technology.

Yates, J. (2016). *Python: Practical Python Programming for Beginners and Experts*
(Beginner Guide)

External links

https://www.w3schools.com/python/python_file_handling.asp
https://www.geeksforgeeks.org/python-if-else/?ref=lbp
https://docs.python.org/3.7/library/functions.html#func-str
https://mkyong.com/python/python-how-to-convert-int-to-string/

0001MG082022