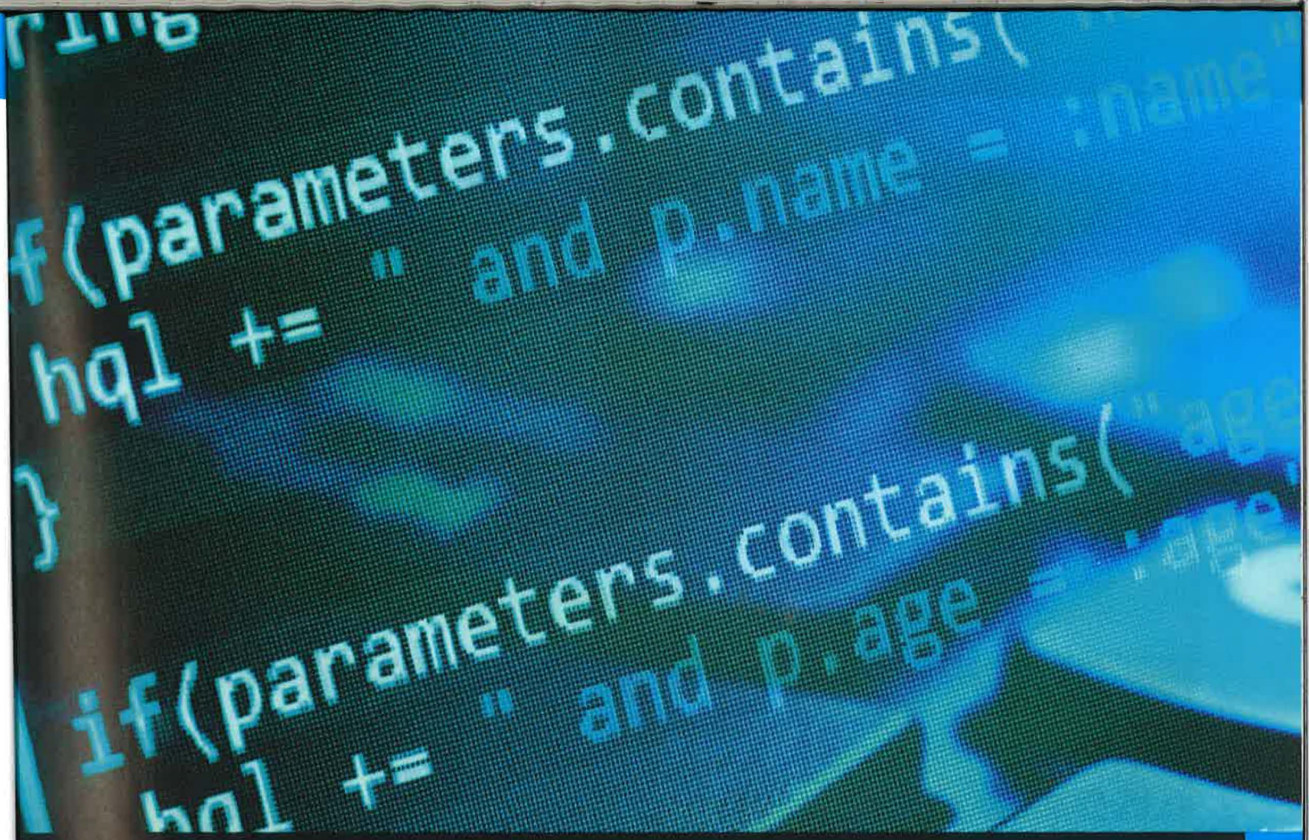
Acknowledgements

Thanks to the following for permission to reproduce images:

Cover image: Soulart/Shutterstock; Chapter opener 1 isak55/Shutterstock; Chapter opener 2 aimy27feb/Shutterstock; Chapter opener 3 Image Source/Getty Images;Chapter opener 4 Devrimb/iStock/Getty Images;Chapter opener 5 Andrew Brookes/Getty Images; Chapter opener 6 Magictorch/Ikon Images/Getty Images;Chapter opener 7 alexaldo/iStock/Getty Images;Chapter opener 8 Ioana Davies (Drutu)/Shutterstock; Chapter openers 9, 10 Kutay Tanir/Photodisc/Getty Images;Chapter opener 11 ILeysen/Shutterstock; Chapter opener 12 Kamil Krawczyk/E+/Getty Images; Chapter opener 13 Aeriform/Getty Images

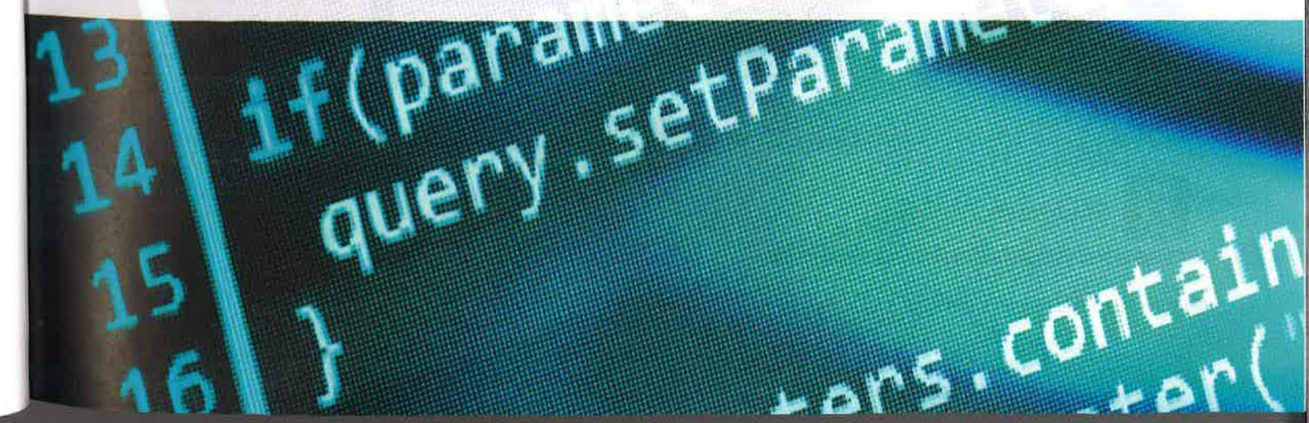


Chapter 1:
Python 3

Learning objectives

By the end of this chapter you will understand how to:

- obtain a simple IDE to support your programming
- use both interactive mode and script mode in Python
- program and save a text-based application in script mode.



1.01 Getting Python 3 and IDLE For Your Computer

Python 3 is the latest version of the Python programming language. It is a loosely typed script language. Loosely typed means that it is usually not necessary to declare variable types; the interpreter looks after this. Script languages do not have a compiler. This means that, in general, Python programs cannot run as quickly as compiled languages; however, this brings numerous advantages, such as fast and agile development.

Python is a powerful, modern programming language used by many famous organisations such as YouTube and National Aeronautics and Space Administration (NASA) and it is one of the three programming languages that can be used to develop Google Apps.

There are installers for Windows and Apple computers available at https://www.python.org/downloads/. You should choose the latest stable version of Python 3 (Python 3.5.0 at time of writing). If you have a Raspberry Pi, then two versions of Python are already installed.

On the Raspberry Pi you can start programming in **interactive mode** straight away by selecting *Python 3* from *Programming* in the main *Menu* in the task bar (Figure 1.01).

Figure 1.01 Starting Python 3 on a Raspberry Pi

This opens IDLE which is the **IDE** (Integrated Development Environment) that comes packaged with Python (Figure 1.02).

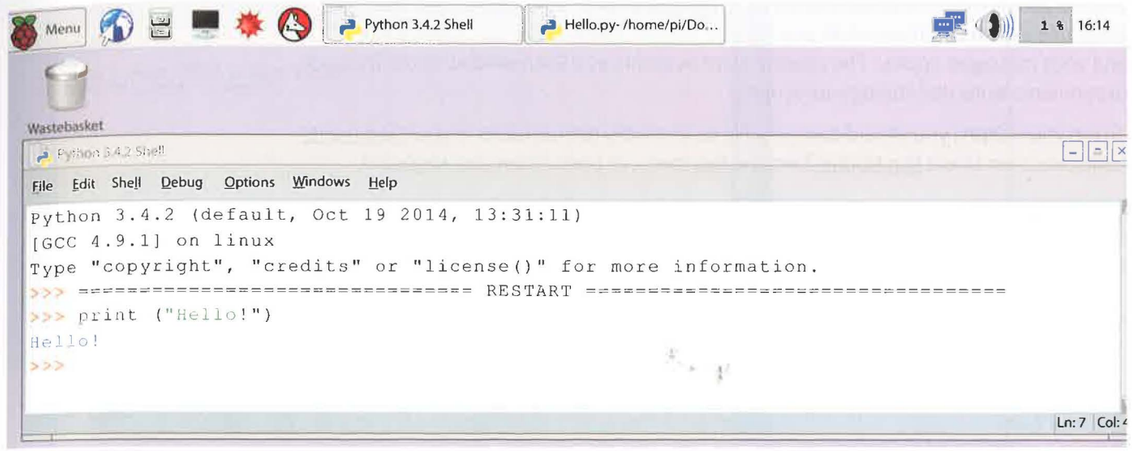


Figure 1.02 IDLE's Python Shell – working in interactive mode on a Raspberry Pi

KEY TERMS

Interactive mode: When writing and running code in the Python Shell window, interactive mode allows us to try out snippets of code without saving.

IDE: An Interactive Development Environment is a special text editor with useful built-in tools for programmers.

After installing Python 3 on Apple computers, IDLE can be found in the main *Python 3* folder in your *Applications* folder.

On Windows computers, once installed, IDLE can be opened by looking for the *Python 3.5* folder found in *All Programs* when opening the *Start* menu. From the *Python 3.5* folder choose *IDLE*.

In all cases this opens a window containing the Python Shell. This can run simple programs at the >>> prompt. Executing small programs in the Shell window is referred to as working in interactive mode. It provides a very useful environment for running short code experiments when developing larger programs in **script mode**. Throughout this book you will be prompted to try out code snippets and run short experiments so that you get used to new functions and syntax in interactive sessions. These sessions can be accessed extremely quickly by opening IDLE and typing directly into the Shell window.

KEY TERMS

Script mode: When writing code in a new window in IDLE that will be saved in a file as a Python script.

To create a script that can contain more complex programs and, more significantly, can be saved and reused, you should obtain a new window by selecting *New File* from the *File* menu. This opens a blank script window into which you can type and save your code (always with

the extension .py). IDLE provides help with code colouring and auto-indenting in whichever mode you are working.

In script mode, the Shell window takes on a new role as a console. Text output from your programs appears in this window (see Figure 1.03). It is also where users provide input, and error messages appear. The console is still available as a Shell window to use for quick experiments while developing your scripts.

To run your scripts, you should save your file to a sensibly named folder in your *Documents* folder and then select *Run Module* from the *Run* menu, or press F5 on your keyboard.

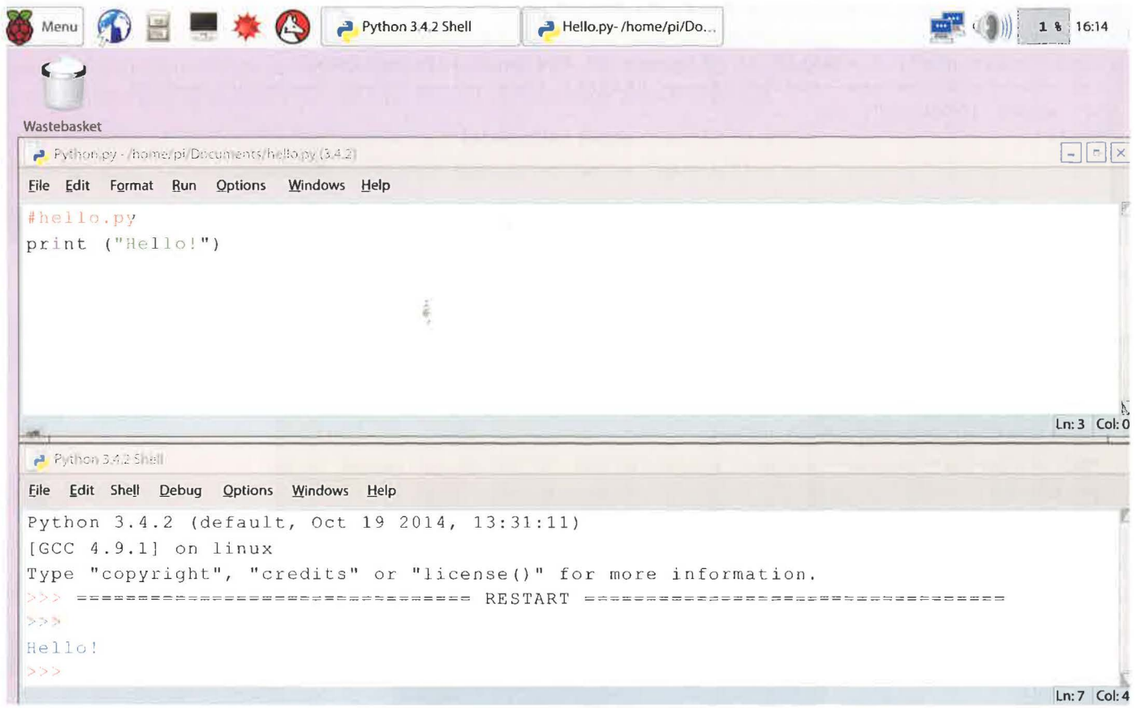


Figure 1.03 IDLE's Python Shell and a script window open on a Raspberry Pi

1.02 Other Integrated Development Environments

IDLE is perfectly adequate for performing all the tasks required in this book. However, if you have been programming with IDLE for a number of years, you might like to try one of the other many IDEs available.

The one that is used for the remainder of the screenshots in this chapter, and occasionally later in the book, is Wing IDE 101 (Figure 1.04). This is a free version of a commercial IDE that provides a carefully selected set of facilities that are useful for students. It can be downloaded from http://wingware.com/downloads/wingide-101 where brief introductory videos and installation instructions are available. Please be aware that the Raspberry Pi is currently not powerful enough to run this or most other commercial IDEs satisfactorily. Wing IDE 101 is available for Windows, Apple computers, Ubuntu and other versions of Linux.

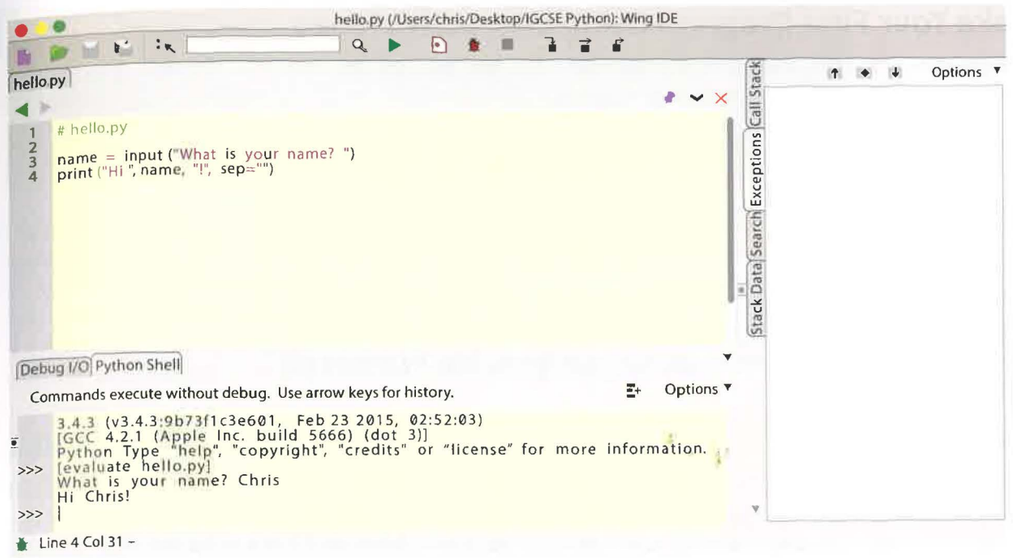


Figure 1.04 Wing IDE 101 Integrated Development Environment.

The large panel in the middle of the application is where you write your scripts. Interactive sessions can be run in the Shell tab below this window.

There are two ways to run a program in Wing IDE. Clicking the run button (▶) will access the Python Shell as shown in Figure 1.04. An alternative - and recommended – way of running your scripts is to click on the bug (🐞) to the right of the run button (Figure 1.05) This opens the Debug I/O panel and now provides error messages in the Exceptions tab on the right.

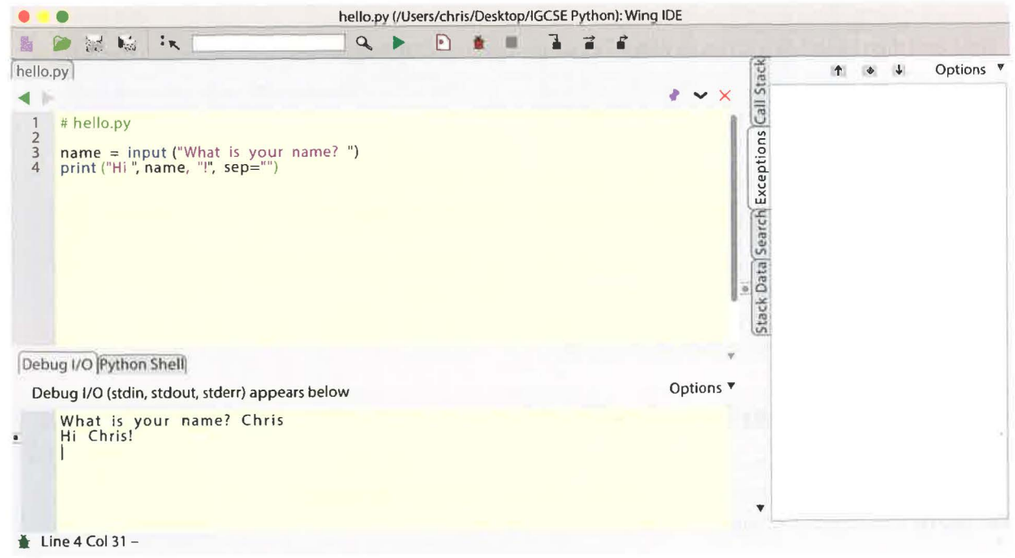


Figure 1.05 Wing IDE 101 showing input and output after pressing the bug button

1.03 Make Your First Program Using Interactive Mode

In IDLE's interactive mode window or in the Python Shell tab in Wing IDE, type out the following line of code at the >>> prompt and then press return.

INTERACTIVE SESSION

```
>>> print('Hello world!')
```

You have now run your first interactive mode program. Your code told the computer to print the text 'Hello world!' to the screen. It executed your code when you pressed the return key on your keyboard. You can also use interactive mode as a simple calculator. Try entering this sum and press return:

INTERACTIVE SESSION

```
>>> 3*4
```

TIP

Interactive sessions are used to illustrate simple concepts or to show the correct use of some new syntax. It is a good idea to start your own interactive session and try the code yourself. You may well want to experiment further to deepen your understanding.

1.04 Make Your First Program Using Script Mode

Working in IDLE select *New File* from the *File* menu to open a new window into which you can type your code and then save it as a script. In Wing IDE, simply type into the main script panel. Whichever IDE you are using, copy the following code and then save your file as `hello.py` to a new folder called *Python Code* in your *Documents* folder.

```
# hello.py
print('Hello world!')
```

If using IDLE, run the script by selecting *Run Module* from the *Run* menu or by pressing F5. In Wing IDE click the bug button.

Any code preceded by a hash symbol (#) is called a comment. This is ignored by the computer when executing the script and is purely for the programmer. It can be useful to include the file name in its own comment at the top of a script.

1.05 Graphical user interface Applications

Although not required by the syllabus, your Python scripts are not limited to text-based applications. By importing the **tkinter** module, it is easy to produce visually rich **graphical user interfaces** (**GUIs**) and attach your algorithms to buttons in windows.

Producing GUI based applications is outside the syllabus.

KEY TERMS

tkinter: An example of a GUI toolkit which is provided as part of the standard library when you install Python.

graphical user interface (GUI): Graphical user interfaces contain items like buttons, text entry boxes and radio buttons.

Chapter 5, GUI applications, is an optional chapter included in this book. In it, you will learn how to build your own GUIs and how to repurpose your algorithm solutions to work with them. From Chapter 5 onwards, there will be some tasks provided that include making GUIs. Although these are not required by the Cambridge IGCSE and O Level Computer Science syllabus, repurposing your solutions to work with GUIs will make you a more flexible programmer and allow you to produce more professional looking applications.

1.06 Additional Support

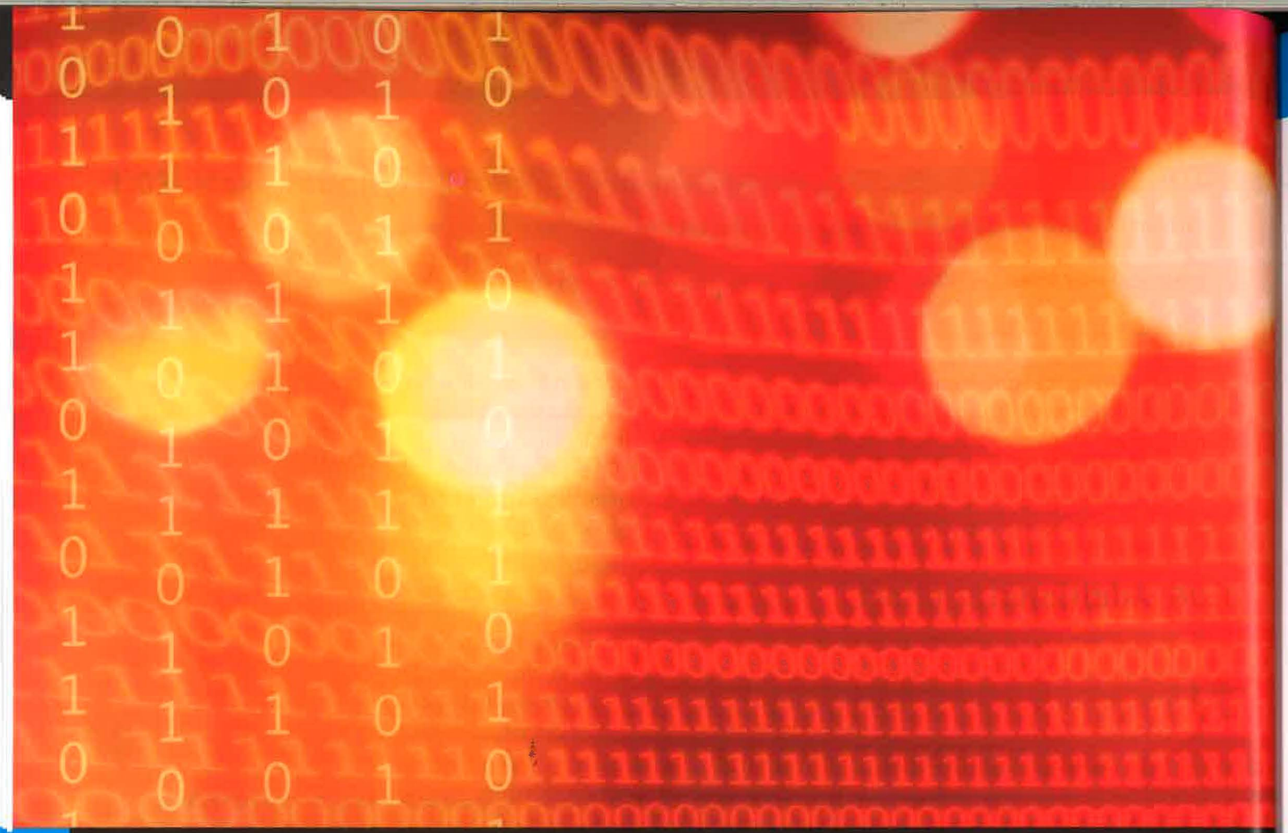
The intention of this book is to introduce programming concepts that make use of the non-language-specific formats included in the syllabus. Python 3 is used to provide the opportunity for you to use a real programming language to develop your understanding of these concepts. The official documentation for the Python programming language can be accessed at https://docs.python.org/3/.

A simple syntax reference guide that can be printed out and fits in your pocket is available from the Coding Club website at http://codingclub.co.uk/codecards/.

Further resources can be downloaded from cambridge.org/9781316617823.

Summary

- Python 3 is a loosely typed programming language that is designed to encourage easily read code.
- Python 3 comes with a simple Integrated Development Environment called IDLE.
- There are many other IDEs available, such as Wing IDE 101, which is specifically designed for students.
- There are three main styles of programming in Python 3:
  - interactive mode: quick tests and trials that can be programmed in the Python Shell
  - text-based: in script mode, text-based scripts can be saved so that your applications can be reused
  - GUI applications: full, visually rich applications that can be produced in script mode.

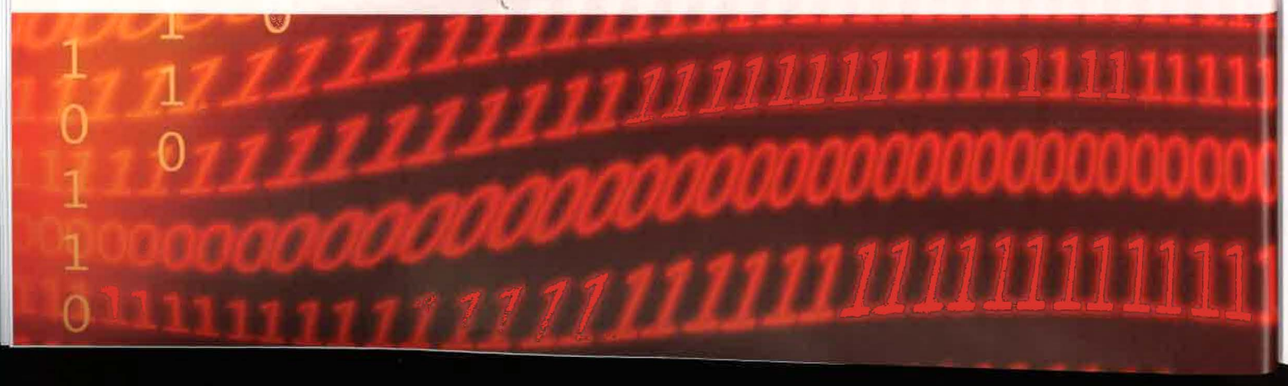


Chapter 2: Sequence

Learning objectives

By the end of this chapter you will:

- know the difference between the three programming constructs: sequence, selection and iteration
- understand the role of flowcharts and pseudocode when designing programs
- understand the main symbols used in flowcharts
- understand the preferred format of pseudocode when using sequence solutions.



2.01 Logical Design Considerations

When designing programs, it is crucial to consider the order in which the task needs to be completed. All tasks will follow some logical order. When working on a solution to a problem, you should first apply the **top-down design** technique to break down the big problem into smaller ones. In terms of computational thinking, this is referred to as **decomposition**.

KEY TERMS

Top-down design: Design process where a complex task is broken down into smaller tasks.
Decomposition: The process of breaking down a large task into smaller tasks.

For example, to calculate the time it would take to complete a journey, you need to know the distance to be travelled and the intended speed. The first step would be to calculate the distance to be travelled. Without this **data** the rest of the task could not be completed.

The **sequence** in which instructions are programmed can be crucial. Consider the following algorithm:

```
Distance = Speed * Time
Speed = 12 kilometres per hour
Time = 15 minutes
```

KEY TERMS

Data: Raw facts and figures.
Sequence: Code is executed in the order it is written.

A human would recognise that the values for speed and time have been given after the calculation. A coded program would simply try to complete the task in the order given and crash. This is because at the time of the calculation no values have been provided for speed or time. In fact, the variables Speed and Time will not even be recognised by the program at this first step.

A human would probably also recognise that the speed is quoted 'per hour' while the time is given in minutes. They would be able to correctly calculate the distance as 3 kilometres (12 * 15/60). Even if the values had been provided before the calculation, the program would calculate distance incorrectly as 180 kilometres by simply multiplying the given values (12 * 15).

2.02 Programming Constructs

Python and other procedural languages make use of three basic programming constructs: sequence, **selection** and **iteration**. Combining these constructs provides a programmer with the tools required to solve logical problems. Selection and iteration offer a number of alternative approaches and are covered in detail in Chapters 6 and 7.

KEY TERMS

Selection: Code branches and follows a different sequence based on decisions made by the program.
Iteration: Code repeats a certain sequence of code a number of times depending on certain conditions.

Sequence

The order in which a process is completed is often crucial. Take the mathematical expression A + B × C + D. The rules of precedence state that the multiply operation must be completed first. If a programmer wishes that the operations A + B and C + D be completed before multiplying, then it would be necessary to either complete the two additions separately first or write the expression in the form (A + B) × (C + D).

In programming, the sequence is indicated by the order in which the code is written, usually top to bottom. The program will execute the first line of code before moving to the second and subsequent lines.

Sequence is the subject of this chapter so this will be discussed in more detail later.

Selection

Often your programs will perform different processes dependent on user input. Consider a system designed to provide access to the school network based on when a user inputs a username and password. The system would need to follow a different path if the user inputs an incorrect password or username. In this circumstance, the user might simply be prompted to re-input their details. See Chapter 6 for more details.

Iteration

It is common for a program to perform identical processes on different data items. Consider a program that takes a series of coordinates and produces a line graph. The code that provides the instructions that plot each new coordinate will be repeated for each of the coordinates given. To repeat instructions, we put them in a loop, which is referred to as iteration. See Chapter 7 for more details.

2.03 Design Tools

When you design programs, it is normal to plan the logic of the program before you start to code the solution. This is an important step in the design of effective systems because a flaw in the logic will often result in programs that run but produce unexpected outputs.

The first step in the design process is to break down the problem into smaller problems. This is called top-down design. It makes it easier to plan and write code for these smaller problems. A **structure diagram** is used to help organise the top-down design. Chapter 8 provides more detail about top-down design and structure diagrams.

The next stage is to design an algorithm for the individual problems. Two approaches that can be used at this stage to help generate logically accurate systems are **flowcharts** and **pseudocode**.

KEY TERMS

Structure diagrams: A diagrammatical method of expressing a system as a series of subsystems.
Flowchart: A graphical representation of the logic of a system.
Pseudocode: A language-independent system for defining the logic of a system without the need for strict syntax.

To succeed in your course, you will be expected to have a working understanding of flowcharts and pseudocode. You will need to be able to use them to explain the logic of your solutions to given tasks. Both methods are used throughout this book.

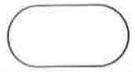
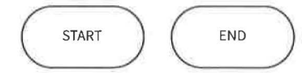
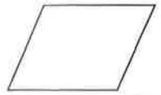
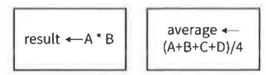
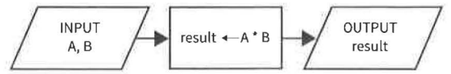
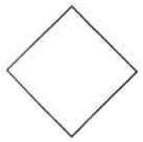
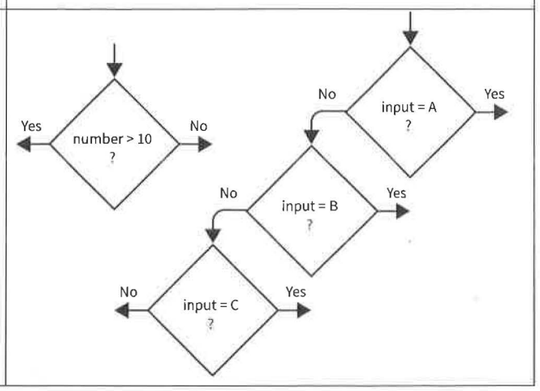
SYLLABUS CHECK

Problem solving and design: use flowcharts and pseudocode.

2.04 Flowcharts

Flowcharts are graphical representations of the logic of the intended system. They make use of symbols to represent operations or processes and are joined by lines indicating the sequence of operations. Table 2.01 details the symbols used.

Table 2.01

Symbol	Notes	Example
Terminator 	The START or END of a system.	
Input or output 	Use when INPUT is required from the user or OUTPUT is being sent to the user.	
Process 	A process within the system. Beware of making the process too generic. For example, a process entitled 'Calculate Average' would be too generic. It needs to indicate the values used to calculate the average.	
Data flow line	Joins two operations. The arrowhead indicates the direction of the flow. Iteration (looping) can be indicated by arrows returning to an earlier process in the flowchart.	
Decision 	A point in the sequence where alternative paths can be taken The condition is written within the symbol. Where multiple alternatives exist, this is indicated by chained decision symbols. Each 'No' condition directs to another decision in the process.	

2.05 Pseudocode

Pseudocode is a method of describing the logic and sequence of a system. It uses keywords and constructs similar to those used in programming languages but without the strict use of syntax required by formal languages. It allows the logic of the system to be defined in a language-independent format. This can then be coded using any programming language. Hence, the flow diagrams and pseudocode in this book are almost entirely the same as those used in the Visual Basic sister book of the series.

Pseudocode follows a number of underlying principles:

- Use capital letters for keywords close to those used in programming languages.
- Use lower case letters for natural language descriptions.
- Use indentation to show the start and end of blocks of code statements, primarily when using selection and iteration.

One of the advantages of learning to program using Python is that the actual coding language is structured in a similar way to natural language and therefore closely resembles pseudocode. Python IDEs such as IDLE or Wing IDE also automatically indent instructions where appropriate.

SYLLABUS CHECK

Pseudocode: understand and use pseudocode for assignment, using ←.

2.06 Pseudocode Example

The following pseudocode is for an algorithm that accepts the input of two numbers. These values are added together and the result is stored in a memory area called `answer`. The value in `answer` is then displayed to the user. (In Chapter 3, we will learn that this memory area is known as a variable.)

```
INPUT number1
INPUT number2
answer ← number1 + number2
OUTPUT answer
```

Note the use of ← to show the passing of values. This is pseudocode's assignment operator. In pseudocode the equals symbol (=) is used to compare values. It is important to note that in Python the equals symbol is used for assignment and two equals symbols (==) are used to compare values.

TASK

Task 1

Construct a flowchart to represent this pseudocode example.

2.07 Effective use of Flowcharts and Pseudocode

Due to their universal nature, flowcharts and pseudocode are used extensively in the Cambridge IGCSE and O Level Computer Science syllabus.

The aim of this book is to help you to learn how to code effective systems in Python. The following chapters make use of flowcharts and pseudocode to define the logic of systems before moving on to specific Python solutions.

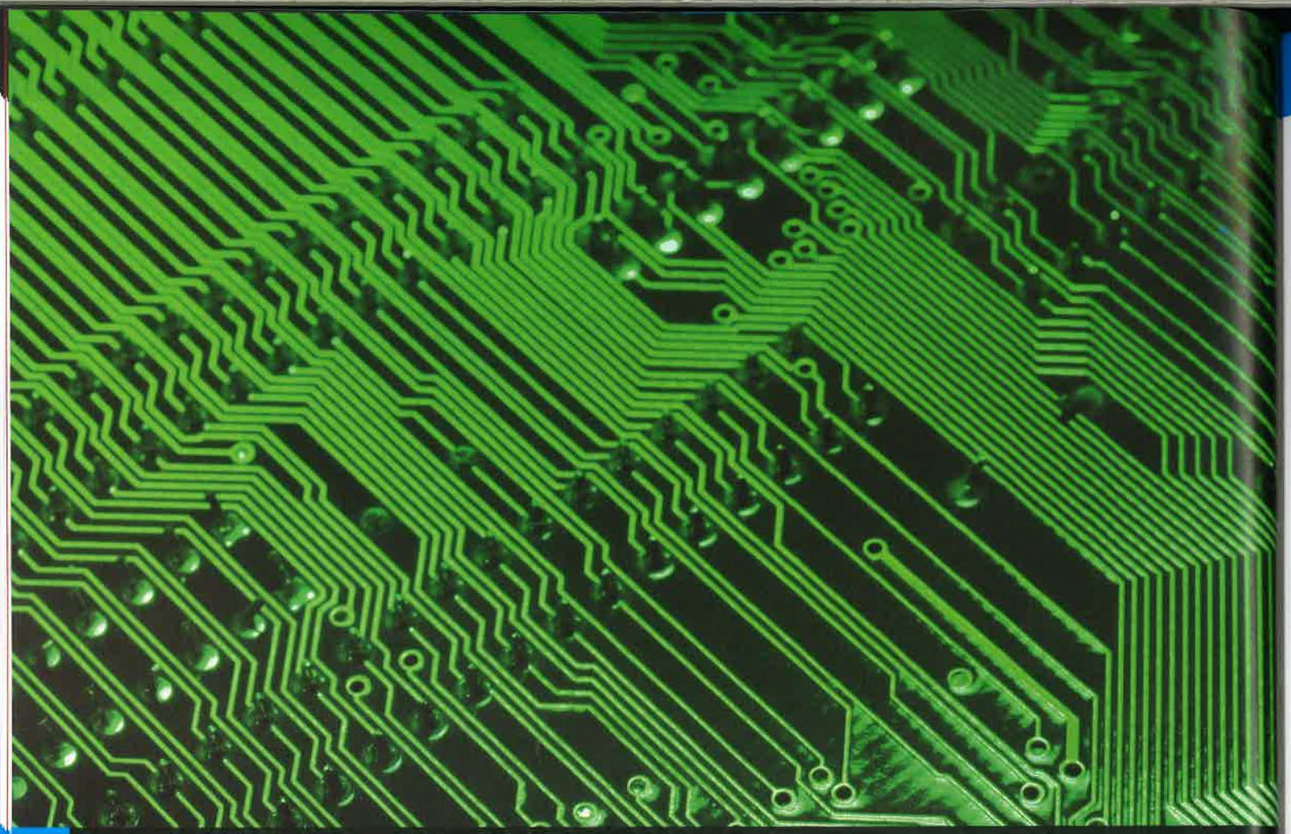
TIP

After completing a flowchart or pseudocode, it is a good idea to try and follow it through a step at a time in the same way a computer would in order to identify if you have any missing steps.

Learning how to explain the logic of programs by using these design techniques is important not only in your preparation for examination but also for your preparation in using the languages of the future. Language syntax is likely to change but the need for effective computational thinking will remain.

Summary

- Programmers make use of three constructs when writing code:
  - sequence: the logical order in which code is executed
  - selection: branching of code onto different paths based on certain conditions
  - iteration: repetition of sections of code.
- Before coding a program, it is crucial to design an appropriate algorithm.
- Flowcharts are graphical representations of the logic of a system. They make use of symbols to represent operations or processes, and lines indicate the sequence of operations.
- Pseudocode describes the logic of a system in a similar way to a programming language but without such strict syntax requirements.

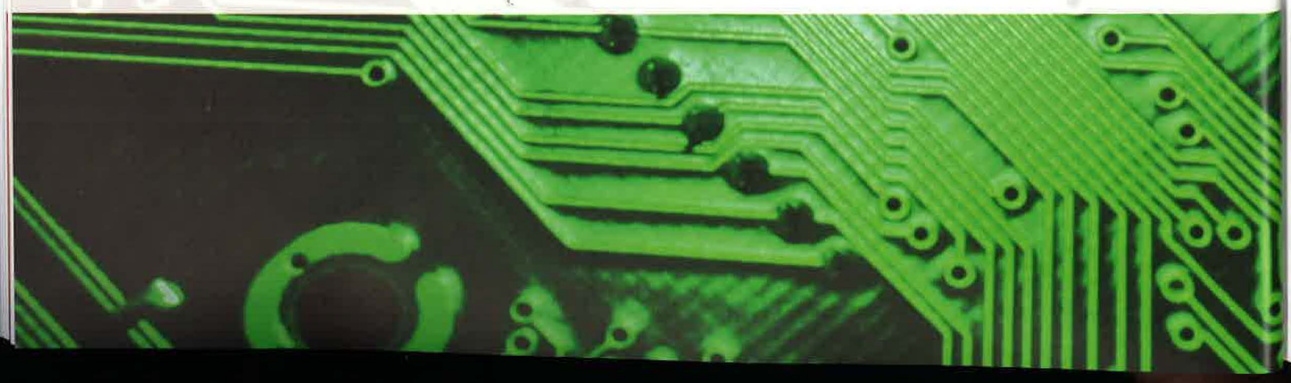


Chapter 3:
Variables and Arithmetic Operators

Learning objectives

By the end of this chapter you will understand how to:

- declare and use variables and constants
- use the data types Integer, Real, Char, String and Boolean
- use basic mathematical operators to process input values
- design and represent simple programs using flowcharts and pseudocode.



3.01 Variables and Constants

Programs are normally designed to accept and input data, and process that data to produce the required output. Data used in programs can vary depending on the aim of the program; a calculator will process numerical data while a program designed to check email addresses will process textual data. When writing programs, you will use variables or constants to refer to these data values. A **variable** identifies data that can be changed during the execution of a program while a **constant** is used for data values that remain fixed. In many computer languages, the **data type** must be provided when a variable or constant is declared. The data type is used by the computer to allocate a suitable location in memory. These languages, such as Java, are said to be strongly typed.

KEY TERMS

Variable: The identifier (name) given to a memory location used to store data; the value can be changed during program execution.

Constant: A named memory location that contains data that can be read but not changed by the program. (In Python, the data can be changed. However, by capitalising your variable name, you are indicating to readers of your code the intention that the value of the data should not be.)

Data type: The format of the data in the field.

Python is an example of a loosely typed programming language. In Python, all variables are in actual fact objects. The computer decides on a variable's data type from the context you provide. Compare these two variable declarations, first in Visual Basic and then in Python:

In Visual Basic:

```
Dim Score As Integer = 0
```

In Python:

```
score = 0
```

3.02 Types of Data

If Python decides which data types are required for the programmer, how can we know what data type has been allocated? This is achieved by using the built in `type()` function. Study this interactive session in the Python Shell to see how to use this function:

INTERACTIVE SESSION

```
>>> my_integer = 3
>>> type(my_integer)
<class 'int'>
>>> my_string = 'hello'
>>> type(my_string)
<class 'str'>
```

The basic data types you need to know are identified in Table 3.01.

Table 3.01

Data type	Description and use	Python `type(variable)` query returns:
Integer	Whole numbers, either positive or negative Used with quantities such as the number of students at a school – you cannot have half a student.	`'int'`
Real	Positive or negative fractional values Used with numerical values that require decimal parts, such as currency. Real is the data type used by many programming languages and is also referred to in the Cambridge IGCSE and O Level Computer Science syllabus.	`'float'` Python does not use the term Real. The equivalent data type in Python is called 'floating point'.

Char	A single character or symbol (for example A, z, $, 6)
	A Char variable that holds a digit, cannot be used in calculations.
		`'str'`
		Python treats characters as small Strings.
		Note:
		`>>> my_var = '3'`
		`>>> type(my_var)`
		`<class 'str'>`
		`>>> my_var = 3`
		`>>> type(my_var)`
		`<class 'int'>`

String	More than one character (a String of characters) Used to hold words, names or sentences.	`'str'`

Boolean	One of two values, either True or False
	Used to indicate the result of a condition. For example, in a computer game, a Boolean variable might be used to store whether a player has chosen to have the sound effects on.
		`'bool'`
		e.g.
		`>>> sfx = False`
		`>>> type(sfx)`
		`<class 'bool'>`

SYLLABUS CHECK

Programming concepts: understand and use Integer, Real, Char, String and Boolean.

3.03 Pseudo Numbers

Telephone numbers and ISBNs both consist of digits but are not truly numbers. They are only a collection of digits used to uniquely identify an item; sometimes they contain spaces or start with a zero. They are not intended to be used in calculations. These are known as pseudo numbers and it is normal to store then in a String variable. If you store a mobile phone number as an Integer, any leading zeroes will be removed and spaces and symbols are not permitted.

3.04 Naming Conventions in Python

There are a variety of naming conventions in Python. Here are a few of them:

Variable Names

Use all lower case, starting with a letter and joining words with underscores. It is considered good practice to use descriptive names as this aids readability and reduces the need for so much commenting.

For example:

```
score_total = 56    ✓
Total = 56          ✗
t = 56              ✗
```

FURTHER INFORMATION

There are 31 reserved words that cannot be used as your own variable names:

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

Constants

Use all upper case characters to indicate constants.

For example:

```
PI = 3.1415
```

It is considered good practice to give global variables an initial value when **declaring variables**: this is known as **initialising variables**. See the next section for more about global and local variables.

KEY TERMS

Declaring variables: When a variable is given a name and assigned no value. It is important to declare or initialise global variables.

Initialising variables: When a variable is given a start value.

3.05 Variable Scope

When declaring a variable, the placement of the declaration in the code will determine which elements of the program are able to make use of the variable.

Variable Scope is outside the syllabus

Global variables are those that can be accessed from any routine within the program. To give a variable global status, it must be declared outside any specific subroutine. It is good practice to make all the global variable declarations at the start of your code directly below any import statements.

To access global variables in functions, they can be called as normal; however, if the function is going to change the value stored in the global variable it must be re-declared using the `global` keyword (see example on page 18).

Local variables can only be accessed in the code element in which they are declared. They are used when the use of the variable will be limited to a single routine such as a function. Using local variables reduces the possibility of accidentally changing variable values in other parts of your program.

KEY TERMS

Global variables: Variables that can be accessed from any routine within the program.

Local variables: Variables that can only be accessed in the code element in which they are declared.

In the following Python code example, there is a global variable (`player_score`) and one local variable (`result`) As the value of the global variable might be changed by the `update_player_score()` function, `player_score` needs to be re-declared at the start of the function with the `global` keyword:

```
player_score = 0

def update_player_score():
    global player_score
    result = 5
    if player_score < result:
        player_score = player_score+1
```

3.06 Arithmetic Operators

There are a number of operations that can be performed on numerical data. Combining these operations and appropriate variables allows you to create programs that are capable of performing numerical computational tasks.

The basic operators used in Python 3 are shown in Table 3.02

Table 3.02

Operation	Example of use	Description
Addition	`result = number1 + number2`	Adds the values held in the variables `number1` and `number2` and stores the result in the variable `result`.
Subtraction	`result = number1 - number2`	Subtracts the value held in `number2` from the value in `number1` and stores the result in the variable `result`.
Multiplication	`result = number1 * number2`	Multiplies the values held in the variables `number1` and `number2` and stores the result in the variable `result`.
Division	`result = number1 / number2`	Divides the value held in the variable `number1` by the value held in `number2` and stores the result in the variable `result`.
Integer division	`result = number1 // number2`	Finds the number of times `number2` can go into `number1` completely, discards the remainder, and stores the result in the variable `result`.

TIP

If you find yourself having to write some Python 2 programs, it is important to be aware that the syntax for division and Integer division is the other way around.

Now is a good time to open up a Python Shell and have an interactive session to try out some of these operators yourself. To get you started, try completing these two interactive sessions by pressing return after the final line in each case. Don't forget to find out what value is stored in `c` as well.

INTERACTIVE SESSION

```
>>> a = 7
>>> b = 3
>>> c = a/b
>>> type(c)
```

TASK

Task 1

How can you find out what value is stored in `c`?

INTERACTIVE SESSION

```
>>> a = 7
>>> b = 3
>>> c = a//b
>>> type(c)
```

3.07 Programming Tasks

DEMO TASK

Multiply Machine

Produce a system called Multiply Machine that takes two numbers inputted by the user. It then multiplies them together and outputs the result.

TIP

Whenever you are provided with a demo task, it is a good idea to open a new file in script mode and copy in the code provided. Think about what each line of code is doing as you type. Then save the script and try it out.

First you need to design the algorithm. Figure 3.01 shows flowchart and pseudocode solutions for the task.

EXTENSION TASKS

Task 5

Program a system that takes as inputs:

- The length of the base of a triangle.
- The perpendicular height of the triangle.

The system should output the area of the triangle.

Task 6

Program a system that takes as inputs:

- The average speed of a car over the length of a journey.
- The distance that the car has to travel.

The system should output, in minutes, the length of time the journey will take.

Task 7

Program a system that takes the three inputs required to calculate the area of a trapezoid and outputs the area.

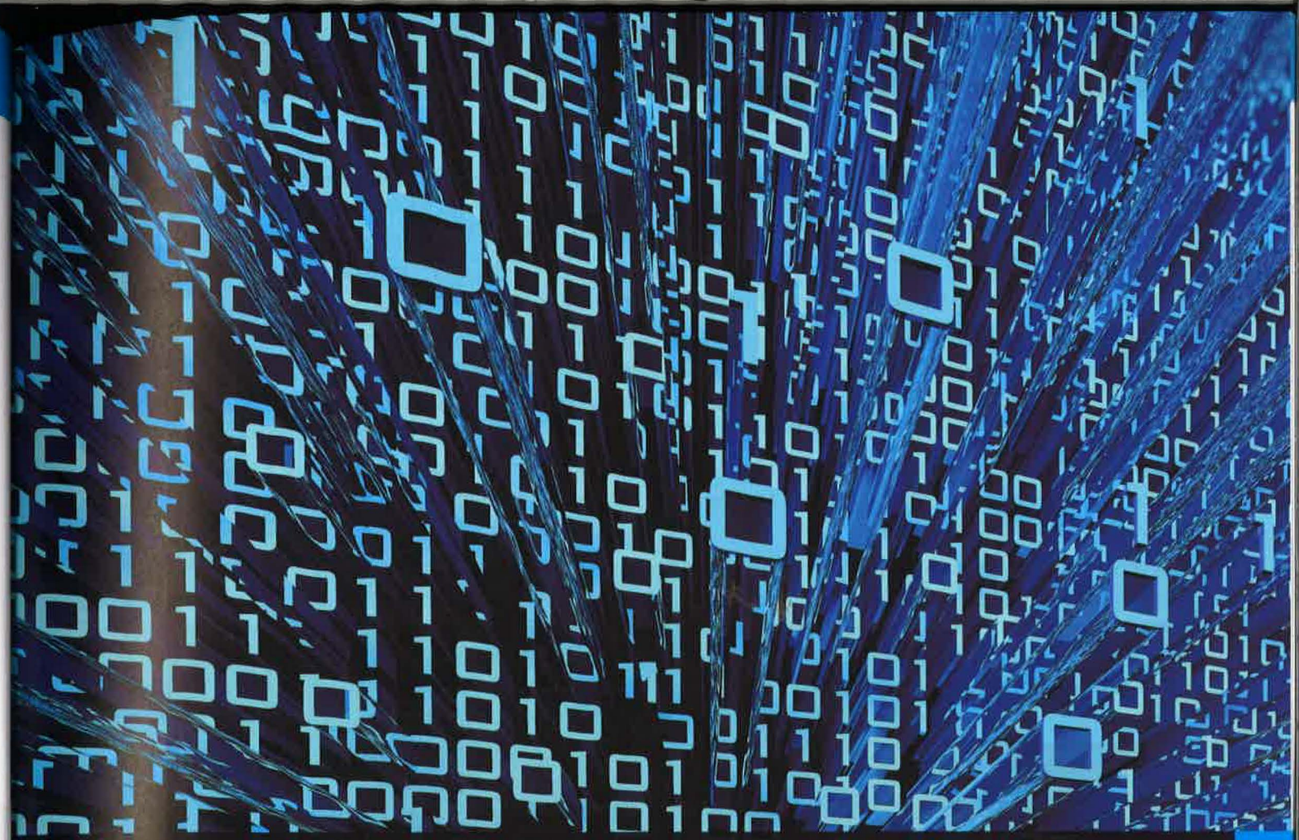
Task 8

Program a system that takes the length of one side of a regular octagon and outputs the resultant area of the octagon.

Hint: To take a square root of a number in Python use this code: `number**0.5`

Summary

- Programs use variables and constants to hold values.
- Variables and constants have identifiers (names) which are used to refer to them in the program.
- Variables are able to have the value they contain changed during the execution of a program. The values within constants remain the same while the program is running.
- In Python, variable names should be descriptive and consist of lower case words joined by underscores.
- In Python, constant names should contain all capital letters. In Cambridge IGCSE and O Level Computer Science pseudocode, they should be preceded with the CONSTANT keyword.
- It is important to know what data types your variables are using. This can be checked by using the `type()` function in Python.
- The `input()` function returns values from the user as String data types. If number inputs are required, the values returned must be cast into Integers or Floats using the `int()` or `float()` functions.
- Mathematical operators can be used with values held in numeric variables.
- Local variables are those that are declared inside a subroutine (see Chapter 4). They cannot be accessed by the rest of the program.
- Global variables are accessed by all parts of a program and are often initialised near the top of a script.
- When designing algorithms, it is crucial to consider the logical sequence of execution. It is important to declare and initialise global variables as well as obtaining user input before completing any processing that requires them.



Chapter 4: Subroutines

Learning objectives

By the end of this chapter you will understand:

- how subroutines are used in programming
- how values are passed to and received from subroutines
- how to design, program and use a function
- how to design, program and use a procedure.

4.01 Subroutines

A subroutine is a sequence of program code that performs a specific task but does not represent the entire system.

All subroutines in Python require a name and the keyword `def` which is short for define.

When a subroutine is activated (this is referred to as 'called'), the calling program is halted and control is transferred to the subroutine. After the subroutine has completed execution, control is passed back to the calling program. This modularised approach to programming brings with it advantages over a simple sequenced program.

Consider a GUI program that maintains its running status while waiting for various subroutines to be called by activation of event triggers. The subroutines execute their code and pass control back to the main program.

This allows the programmer to generate the complete program from a series of individual subroutines. Some code is executed when the script is loaded, other elements when certain buttons are clicked and possibly further elements of code are activated when text is changed in a text box. Imagine the complexity of the program code if only a single sequence of code was available to the programmer.

SYLLABUS CHECK

Programming concepts: use predefined procedures or functions.

Advantages of Using Subroutines

The ability to call subroutines from the main code offers a number of advantages:

- The subroutine can be called when needed: A single block of code can be used many times in the entire program, avoiding the need for repeating identical code sequences throughout. This improves the modularity of the code, makes it easier to understand and helps in the identification of errors.
- There is only one section of code to debug: If an error is located in a subroutine, only the individual subroutine needs to be debugged. Had the code been repeated throughout the main program, each occurrence would need to be altered.
- There is only one section of code to update: Improvements and extensions of the code are available everywhere the subroutine is called.

Types of Subroutine

Two main types of subroutine exist:

- **Procedures** are small sections of code that can be reused. They do not return a value. In pseudocode, a procedure is named and takes the form:

```
PROCEDURE... ENDPROCEDURE.
```

They are called by using the CALL statement.

- **Functions** are similar to procedures. The difference is that functions have one or more values passed to them and one or more values are returned to the main program after they have completed running. In pseudocode, a function takes the form:

```
FUNCTION(values to be passed in)... ENDFUNCTION.
```

The CALL statement is used to execute the function but the values required must be passed to the function at the same time:

```
CALL my_function(values required by the function)
```

KEY TERMS

Procedure: A small section of code that can run repeatedly from different parts of the program.
Function: A procedure that returns a value.

4.02 Programming a Function

The syntax for defining a function in Python is shown here:

```
def circle(r):
    # code to draw a circle goes here
```

To draw a circle of radius ten in the main part of the program we would write:

```
circle(10)
```

Notice how the radius has been passed to the function at call time and there is no need to use a CALL keyword as is used in Cambridge IGCSE and O Level Computer Science pseudocode; the function name suffices in Python.

Passing Parameters to a Procedure

The passing of parameters can be very useful. For example, a procedure to check network logon details could take the parameters `username` and `password`. Having checked the data against the logon database, it could return `True` or `False` to indicate if the details match records, or `'update password'` if the password has expired. The procedure could be called repeatedly and passed different parameters every time a user attempts to log on.

DEMO TASK

Multiples

A function is required that will be passed an Integer and output the first five multiples of that value.

Programming a Function is outside the syllabus

The pseudocode for this function and its call from the main program are as shown here:

```
FUNCTION multiples(number)
   FOR i = 1 TO 5
      OUTPUT number * i
   NEXT
ENDFUNCTION

CALL multiples(10)
```

This uses a **FOR loop**. FOR loops are introduced more fully in Chapter 7.

KEY TERMS

FOR loop: A type of iteration that will repeat a section of code a known number of times.

TASK

Task 1

Create a pseudocode algorithm for an amended version of the Multiples procedure that accepts two parameters: a number to use as the multiplier and another to indicate the maximum number of multiplications required.

Functions That Return Values to the Calling Routine

Often programmers write functions that are required to produce answers for repetitive tasks and then return those values to the main program. For example, it might be necessary for a program to calculate the circumference of several circles from their radii.

DEMO TASK

Circumference

A function is required that will be passed the radius of a circle and return the circumference.

The pseudocode for this function and its call from the main program are as shown below.

```
FUNCTION circumference(r)
   c ← 2 * 3.142 * r
   RETURN c
ENDFUNCTION

INPUT radius
circumf ← CALL circumference(radius)
OUTPUT circumf
```

Here is a Python implementation:

```
def circumference(r):
    c = 2*3.142*r
    return c

radius = int(input('What is the radius of your circle?'))

circumf = circumference(radius)
print('The circumference of your circle is', circumf)
```

Note how the function is not activated by use of the keyword CALL in Python. The name of the function is used as a variable in an assignment statement. Each time the name is used, the function is executed and the return value placed in the variable or output indicated.

TASK

Task 2

Create a pseudocode algorithm for an amended version of this function that, when passed the radius, returns the area of a circle.

Test that your algorithm works by programming and running the code in Python.

Returning Two Values from a Function

It is easy to return two values in pseudocode:

```
RETURN value1, value2
```

In Python, this is accomplished in the same way. Look at this interactive session to see how this works:

INTERACTIVE SESSION

```
>>> def my_function():
        return 1,2

>>> a,b = my_function()
>>> print(a)
1
>>> print(b)
2
>>>
```

TASK

Task 3

Create a pseudocode and flowchart algorithm for an amended version of the circumference function. When passed the radius, this function returns the area and the circumference of a circle.

Test that your algorithm works by programming and running the code in Python.

4.03 Programming a Procedure

Programming a Procedure is outside the syllabus

The Python code for a procedure is similar to that used for a function. In this case empty brackets are used to show that no parameters are required by the subroutine. See how this works in the interactive session shown below:

INTERACTIVE SESSION

```
>>> def greeting():
        print('Hello', 'Hello', 'Hello')
>>> greeting()
Hello Hello Hello
>>>
```

Notice how the `greeting()` function contains the built-in function, `print()`.

TASK

Task 4

a Create a pseudocode algorithm for a procedure called `dead_end()` that prints out 'I am sorry, you can go no further this way!' This might then be called in a maze game whenever a player reaches the end of a passage.

b Test that your algorithm works by programming the procedure in Python and providing a call to the procedure.

Summary

- Subroutines provide an independent section of code that can be called when needed from another routine while the program is running. In this way subroutines can be used to perform common tasks within a program.
- As an independent section of code, a subroutine is easier to debug, maintain or update than repetitive code within the main program.
- Subroutines are called from another routine. Once they have completed execution they pass control back to the calling routine.
- Subroutines can be passed values known as parameters.
- A procedure is used to separate out repetitive code from the main program.
- A function is a type of subroutine which can receive multiple parameters and return values.

Chapter 5:
GUI Applications (Optional)

Learning objectives

By the end of this chapter you will understand:

- how to produce and save GUI applications
- how to program windowed applications using the built-in tkinter GUI module
- how to add widgets to a GUI application
- how to lay out widgets in an application window
- how to trigger function calls with buttons

5.01 Introduction

In Chapter 1, you were introduced to interactive mode and script-based programming. This optional chapter shows you how to create programs that appear in windows and have features such as buttons and text boxes. The Cambridge IGCSE and O Level Computer Science syllabus does not require that you produce applications with GUIs; however, you may well want to produce more visually interesting and professional looking solutions to problems. Doing so will also make you a more flexible programmer as you reformat your scripts into GUI applications.

Producing GUI based applications is outside the syllabus.

From now on, normal script-based solutions are going to be referred to as text-based solutions and programs that appear in windows are going to be called GUI solutions. After this chapter, you will often be asked to produce two solutions, first a text-based one and then a GUI solution. Producing the GUI programs can be considered optional extensions.

5.02 Make Your First Application in a Window with a Button

By importing the tkinter module, it is easy to produce visually rich GUIs and attach your algorithms to buttons in windows.

Tkinter is an example of a GUI toolkit and is provided as part of the standard library when you install Python. Therefore, you already have access to the objects and methods required to make GUI applications, and you just need to do these extra tasks:

1. Import the *tkinter* module.
2. Create the main tkinter window.
3. Add one or more tkinter **widgets** to your application.
4. Enter the main event loop, which listens to and acts upon events triggered by the user.

KEY TERMS

widget: Interface items such as buttons and text boxes that can be used to build GUIs.

A button can call for a particular action to happen by referring to a function by name after `command=` in the button definition code. To create the application shown in Figure 5.01 copy the code into a new script and save it as `hello-gui.py` into your Python Code folder:

```
# hello-gui.py

# Import everything required from the tkinter module
from tkinter import *

# Function called by clicking my_button:
def change_text():
    my_label.config(text='Hello World')

# Create the main tkinter window
window = Tk()
window.title('My Application')

# Add an empty tkinter label widget and place it in a grid layout
my_label = Label(window, width=25, height=1, text='')
my_label.grid(row=0, column=0)

# Add a tkinter button widget, place it in the grid layout
# and attach the change_text() function
my_button = Button(window, text='Say Hi', width=10, command=change_text)
my_button.grid(row=1, column=0)

# Enter the main event loop
window.mainloop()
```

Figure 5.01 A GUI application with a button

After running the code, press the 'Say Hi' button to see how this small application works. Notice how the button is linked to the `change_text()` function by `command=` in the button definition.

FURTHER INFORMATION

The tkinter module provides classes, objects and methods that you can access and use in your own applications. Tkinter is written using object-oriented programming (OOP), which is beyond the scope of the syllabus. In OOP programs, object methods are accessed using the dot operator. This can be seen above in the `change_text()` function where the `config()` method is applied to the label widget.

If you want to learn more about OOP you might like to work through *Python: Building Big Apps*, a level 3 book in the Coding Club series, or perhaps try *Introduction to Programming with Greenfoot* by Michael Kölling, which teaches Java programming in a very interactive, game-based way. It is worth noting that while the syllabus focuses on solving problems through a top-down design process, discussed in detail in Chapter 8, OOP is a good example of how to solve problems through bottom-up design.

When laying out GUI applications, you can use the `grid()` method, which organises as many cells as you require in your window using a coordinate system. Note how the numbers start from zero in the top left corner of the window:

`row=0, column=0`	`row=0, column=1`	`row=0, column=2`
`row=1, column=0`	`row=1, column=1`	`row=1, column=2`
`row=2, column=0`	`row=2, column=1`	`row=2, column=2`

It is possible to further arrange tkinter widgets by grouping them in frames.

5.03 Other Tkinter Widgets You Can Use in Your Applications

Below are a few other useful widget examples you might want to include in your applications. These code snippets should all be added after `window = Tk()` and above `window.mainloop()` as indicated by the comment in the following recipe for an empty tkinter window:

```
from tkinter import *

window = Tk()
window.title('My Application')

# widget code goes here

window.mainloop()
```

A text entry box with a label:

```
Label(window, text='Name:').grid(row=0, column=0)
my_text_box = Entry(window, width=15)
my_text_box.grid(row=0, column=1)
```

Two frames:

```
frame1 = Frame(window,height=20,width=100,bg='green')
frame1.grid(row=0, column=0)
frame2 = Frame(window,height=20,width=100,bg='red')
frame2.grid(row=1, column=1)
```

A drop-down menu:

```
options = (1,2,3)
my_variable_object = IntVar() # access the value with .get()
my_variable_object.set('choose:')
my_dropdown = OptionMenu(window, my_variable_object, *options)
my_dropdown.grid()
```

TIP

When programming graphical implementations of tasks set in future chapters, remember to consult the Appendix where you will find a full set of recipes for the widgets you will be asked to use.

TASK

Task 1 – Tkinter Widgets

Open a new script and add the code from the empty window recipe on page 33. Save this script and then add the code for the example widgets, one at a time, to see how they appear. Do not worry about your scripts doing anything at this stage.

DEMO TASK

Gender GUI Application

Create a radio button application that gives the user a choice of two radio buttons to indicate their gender. Your application should show how to align tkinter widgets to the left (West) side of a `grid()` cell. It should also demonstrate how to access the value selected in the radio buttons using a tkinter `StringVar()` object and display the choice made (Figure 5.02).

TIP

When building GUI applications, it is good practice to separate the logic from the design. To do this, compartmentalise your algorithm solutions into functions at the top of your script and then build your GUI code at the bottom of your script.

Here is the Python code that demonstrates how to produce a GUI for this very simple one function program.

```
# gender-gui.py

from tkinter import *

# Functions go here:
def change_text():
    my_label.config(text=gender.get())

# GUI code goes here:
# Create the main tkinter window
window = Tk()
window.title('My Application')

# Add an empty tkinter label widget and place it in a grid layout
my_label = Label(window, width=25, height=1, text='')
my_label.grid(row=0, column=0)

# Add a tkinter button widget, place it in the grid layout
# and attach the change_text() function
my_button = Button(window, text='Submit', width=10, command=change_text)
my_button.grid(row=1, column=0)

# Create a tkinter string variable object for the radio buttons
gender = StringVar()
```



```
# Add two radio button widgets
# Use optional sticky argument to align left
radio1 = Radiobutton(window, text='Female', variable=gender, value='female')
radio1.grid(row=2, column=0, sticky=W)
radio1.select() # pre-selects this radio button for the user
radio2 = Radiobutton(window, text='Male', variable=gender, value='male')
radio2.grid(row=3, column=0, sticky=W)

# Enter the main loop event
window.mainloop()
```

Figure 5.02 A GUI application with two radio buttons

TASK

Task 2 – Drop-down Menu

Rewrite the radio button application but replace the radio buttons with a simple drop-down menu.

5.04 Choosing a Text-based or GUI Application

From Chapter 6 onwards, answers to problems are given as either text-based applications or GUI applications and often both.

Text-based programs offer the benefit of more accurately reflecting the programming style of the syllabus and will help to prepare you for the A Level syllabus. Text-based applications do not involve the additional complexity of having to reference tkinter's GUI widgets.

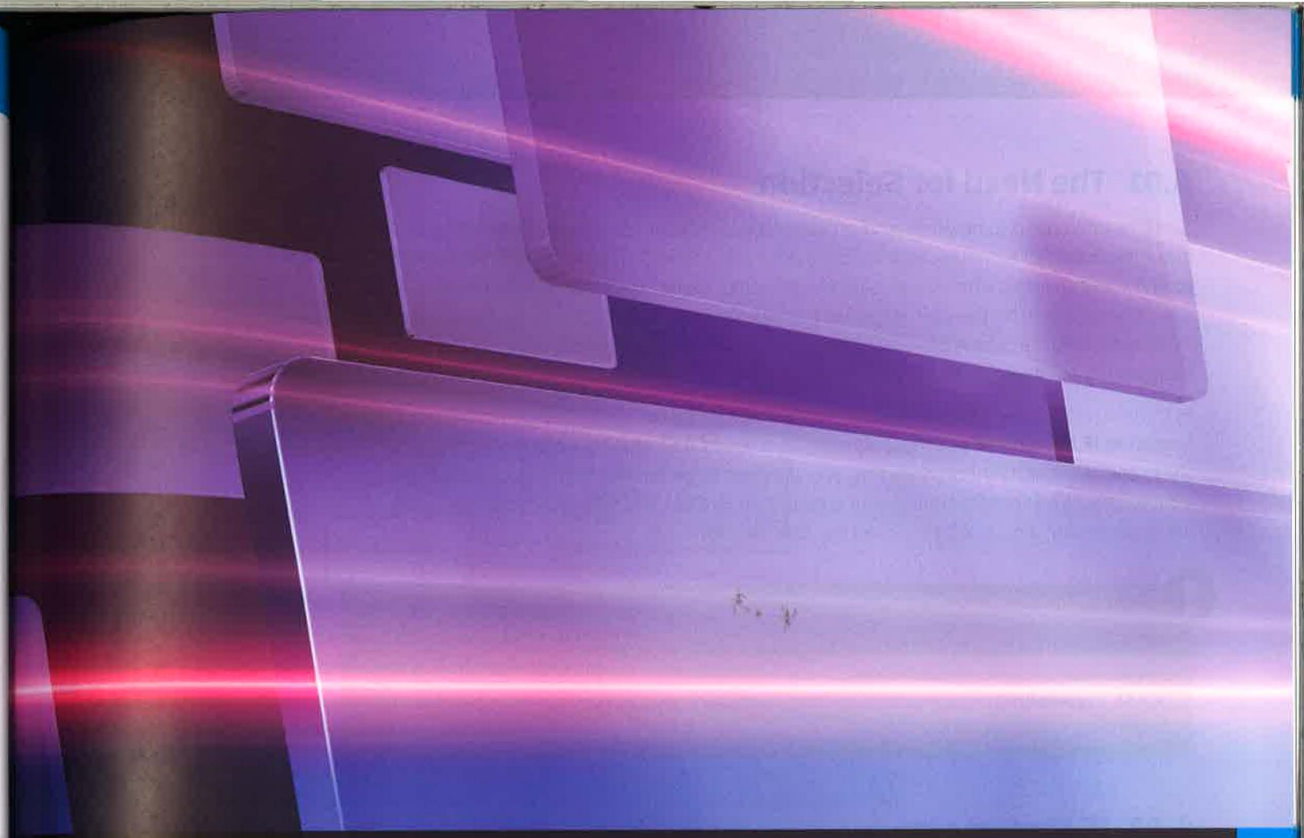
However, GUI applications will offer a richer visual experience and produce systems similar to those commercially available.

It is suggested that you make use of both types of applications. This will best support the development of your computational thinking. A text-based answer is always provided to problems presented in this book in the final chapter, Chapter 14: Solutions. If a GUI-based solution makes sense, these are also provided in the code repository found at the companion website to this textbook at cambridge.org/9781316617823.

Summary

- There are three main styles of programming in Python 3:
  - interactive mode: quick tests and trials that can be programmed in the Python Shell
  - text-based: in script mode, text-based scripts can be saved so that your applications can be reused
  - GUI applications: by importing the tkinter library, full visually rich applications can be produced in script mode.
- GUI programs open in their own window and contain familiar widgets such as buttons and text boxes.
- The tkinter module is a GUI library that is available in the standard Python 3 install.
- Tkinter's grid method allows programmers to position widgets by using cell coordinates starting with (0,0) from the top left corner of a window or frame.
- Text-based programs are all that is required by the syllabus and closely match the logic of the algorithms they implement.
- GUI programs provide a richer experience for the user but introduce added complexity for the programmer.

Producing GUI based applications is outside the syllabus.



Chapter 6: Selection

Learning objectives

By the end of this chapter you will understand:

- how selection can be used to allow a program to follow different paths of execution
- how selection is shown in flowcharts and pseudocode
- the differences between and the advantages of using:
  - IF . . . THEN . . . ELSE . . . ENDIF statements
  - IF . . . THEN . . . ELSE . . . ELSEIF . . . ENDIF statements
  - NESTED IF statements
  - CASE . . . OR . . . OTHERWISE . . . ENDCASE statements
- how to use logical operators when programming selection in algorithms.

6.01 The Need for Selection

Systems often need to be programmed to complete different processes depending on the input received. For example, an automatic door will open if it detects that someone wishes to enter and will shut when no presence is detected. Expert systems provide answers or conclusions based on the user response to previous questions. Systems like these appear to be able to make decisions based on input, but the reality is that the system has been logically designed to complete a certain process based on expected input.

In Python, and many other languages, this is achieved by the use of programming techniques known as **IF statements**. Many languages also have **CASE statements** available to them and, although Python does not, you will need to be familiar with this construct in your exams. When programming your solutions in Python, you will usually replace CASE statements with `if...elif...else` (see Section 6.08).

KEY TERMS

IF statements: A statement that allows the program to follow or ignore a sequence of code depending on the data being processed.

CASE statements: A statement that allows one of several sequences of code to be executed depending on the data being processed.

6.02 IF Statements

If the process for an automatic door was written down, it might appear as: 'If a presence is detected then open the door, otherwise close it.' In simple terms, this is very similar to coding an IF statement. The program will need to provide a condition to evaluate ('If a presence is detected') and two actions will need to be provided depending on whether the outcome is True or False: if True 'open the door', if False 'close the door'.

SYLLABUS CHECK

Pseudocode: understand and use pseudocode, using IF . . . THEN . . . ELSE . . . ENDIF.

In a flowchart, the symbol used to indicate a decision is a diamond. The diamond contains information about the criteria and normally has two exit routes indicating the True and False paths.

The flowchart in Figure 6.01(a) includes the decision symbol. The True and False paths have been indicated; when programmed this will become an IF statement. Once the appropriate action has been performed, the program flow returns back to 'Check for presence at door' and the input is again evaluated by the IF statement.

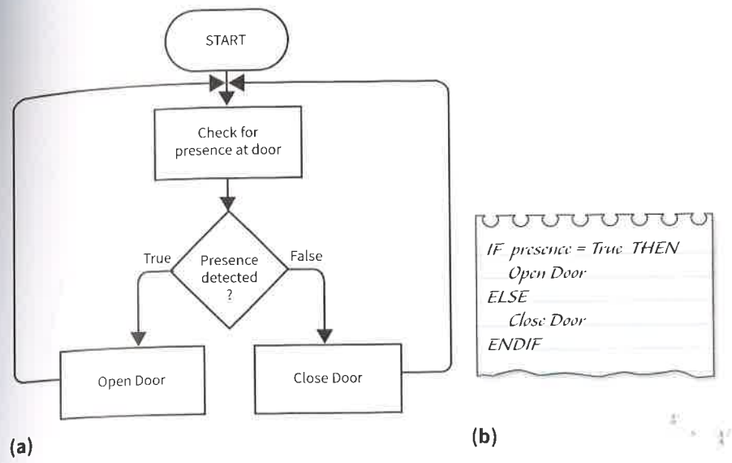


Figure 6.01 Flowchart (a) and pseudocode (b) for a decision

Note that True/False can also be written as Yes/No.

The pseudocode IF statement for the flowchart in Figure 6.01(a) is shown in Figure 6.01(b). The start of the statement is indicated by IF. The condition is written between IF and THEN. The action to be taken if the condition is True follows THEN. The action is indented to improve readability. ELSE indicates any alternative action. Again, the action to be taken should be indented. ENDIF indicates the end of the statement.

Not all IF statements have an alternative action and therefore the ELSE may be omitted. For example, a system used to calculate the cost of a train fare could apply a discount if the passenger is a child. The pseudocode would look like this:

```
discount ← 0.5

INPUT ticket_price

IF ticket for child THEN
    ticket_price ← ticket_price * discount
ENDIF

OUTPUT ticket_price
```

6.03 Logical Operators

In the automatic door example, the only possible inputs were 'detected' or 'not detected'. However, many systems depend on less discrete criteria. An air-conditioning system will receive continuous temperature data and will perform actions based on a variety of temperature data. A system for determining exam grades will calculate the grade output by identifying whether the students' marks fall within certain grade boundaries.

To support the needs of these types of decisions, a number of logical operators exist. They tend to follow the accepted mathematical use of operator symbols. The basic operators supported by Python and their form in pseudocode is shown in Table 6.01

Table 6.01

Operator description	Pseudocode	Python3
is equal to	=	==
is greater than	>	>
is less than	<	<
is greater than or equal to	>=	>=
is less than or equal to	<=	<=
is not equal to	!= or <>	! = <>

The choice of the correct logical operator is important. Using the wrong one can produce unexpected results in your algorithms. Often the way in which the decision required is worded will indicate the appropriate operator to use (Table 6.02).

Table 6.02

Decision in words	Appropriate operator
Apply a discount for students aged under 16	IF student < 16 THEN
Turn on the water cooler when the temperature is 10 °C or more	IF temp >= 10 THEN

Chapter 10 provides you with techniques to identify logical errors such as selecting < when you should have used <=.

6.04 Coding IF Statements in Python

The code for an IF statement in Python is slightly different to that required in pseudocode. The Python programming language is designed to encourage programmers to write easily read code. Indentation of four spaces is required and implemented by IDEs to all blocks of code. Extraneous details sometimes found in other languages such as ending statements with semi-colons as well as a line return; wrapping code blocks in curly brackets as well as indenting and adding ENDIF statements as well as ending the indentation are all dropped. Note though that, unlike in pseudocode, a colon is used before indenting a block of code after, for example, an IF statement. This is to help IDEs to know when to start auto-indenting, and is consistent throughout the language.

DEMO TASK

Sort Two Numbers

Design a system that will take as input two whole numbers. If the second number is larger than the first, the system will output 'Second'; if not the output should be 'First' (Figure 6.02).

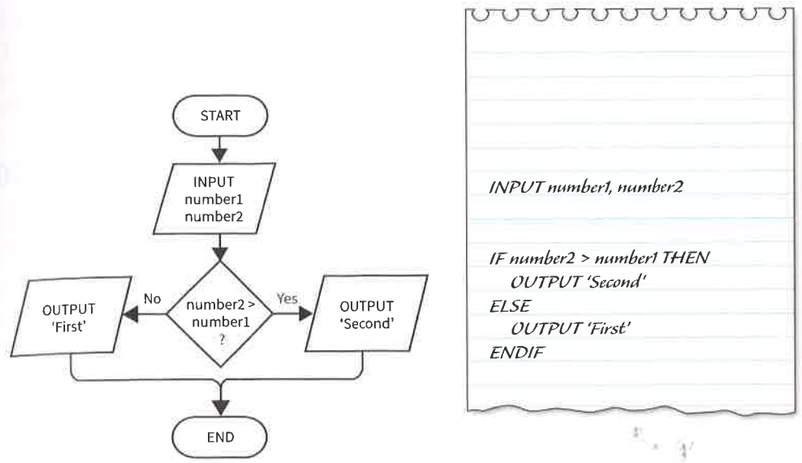


Figure 6.02 Flowchart and pseudocode showing decision

The Python code below implements the system designed in the flowchart and pseudocode shown in Figure 6.02:

```
number1 = int(input('Enter first number: '))
number2 = int(input('Enter second number: '))

if number2 > number1:
    print('Second')
else:
    print('First')
```

TASKS

Task 1

Would the same output be achieved by reversing the condition to `number1 < number2`?

Task 2

The demo task has been reworded very slightly like this:

Design a system that will take as input two whole numbers. If the second number is the same as or larger than the first, the system will output 'Second'; if not the output should be 'First'

a How would this impact on the choice of logical operator?

b Could the condition still be reversed?

6.05 Multiple IF Statements

If the previous example is expanded to provide a third output when the numbers are the same, a simple IF statement would not be sufficient to solve the problem. Problems of this nature can be solved by a series of sequential IF statements, each of which ends before the following one starts as in the example below.

```
INPUT number1
INPUT number2

IF number 2 = number1 THEN
    OUTPUT 'Same'
ENDIF

IF number 2 > number1 THEN
    OUTPUT 'Second'
ENDIF

IF number 2 < number1 THEN
    OUTPUT 'First!'

ENDIF
```

```
number1 = int(input('Enter first number: '))
number2 = int(input('Enter second number: '))

if number2 == number1:
    print('Same')

if number2 > number1:
    print('Second')

if number2 < number1:
    print('First')
```

Although this approach achieves the required outcome, it is inefficient. Consider the situation where both numbers are equal. The first statement would have produced the appropriate output, but even though the conditions in the following IF statements are false, the code must still execute them. The algorithm produces the required output but two IF statements have been executed unnecessarily.

6.06 Nested IF Statements

To avoid the inefficiency of multiple IF statements, it is possible to place one or more IF statements entirely within another. The second and subsequent IF statements will only be executed should the first condition prove to be false. These are known as nested IF statements.

Figure 6.04 shows how a nested IF approach could be applied to the inefficient sequence of IF statements shown in Figure 6.03. Because the second IF statement will only execute if the criteria in the first statement is False, unnecessary execution of IF statements is avoided.

```
INPUT number1
INPUT number2

IF number2 = number1 THEN
    OUTPUT 'Same'

ELSE
    IF number2 > number1 THEN
        OUTPUT 'Second'

    ELSE
        OUTPUT 'First'

    ENDIF

ENDIF
```

```
number1 = int(input('Enter first number: '))
number2 = int(input('Enter second number: '))

if number2 == number1:
    print('Same')

else:
    if number2 > number1:
    print('Second')

    else:
        print('First')
```


Notice how the colon in Python replaces the THEN in the pseudocode and that there is no need to have an ENDIF statement. When writing pseudocode in Cambridge IGCSE and O Level Computer Science papers, it is very important to include ENDIF clauses, make sure you do not forget it.

6.07 CASE Statements

CASE statements are considered an efficient alternative to multiple IF statements in circumstances where many choices depend on the value of a single variable.

Consider the situation where a user must input A, B or C. The code is required to follow different paths depending on which letter has been input. The pseudocode in Figure 6.05(a) shows the approach that would be taken using nested IF statements and Figure 6.05(b) shows a CASE statement.

(a)

```
IF user_input = 'A' THEN
    //Code to follow
ELSE
    IF user_input = 'B' THEN
        //Code to follow
    ELSE
        IF user_input = 'C' THEN
            //Code to follow
        ELSE
            OUTPUT 'Incorrect input'
        ENDIF
    ENDIF
ENDIF
```

(b)

```
CASE user_input
'A': //Code to follow
'B': //Code to follow
'C': //Code to follow
OTHERWISE
    OUTPUT 'Incorrect input'
ENDCASE
```

Figure 6.03 Nested IF (a) and CASE (b) statements

Both approaches achieve the same outcome but the CASE statement is simpler to code and easier to read than the NESTED IF.

However, CASE statements are not available in Python.

6.08 Coding CASE Statements in Python

There are IF statements in Python and there is the possibility to have nested IF statements, but there is no facility to write CASE statements. There is, however, a structure that is easier to read than nested IF statements and more flexible than CASE statements available to Python programmers. Instead of a series of nested IF statements, Python programmers would use `elif` (short for ELSE IF (see section 6.08).)

This is how the pseudocode in Figure 6.03(a) would look in Python:

```
user_input = input('Enter A,B or C: ')

if user_input == 'A':
    # code to follow
elif user_input == 'B':
    # code to follow
elif user_input == 'C':
    # code to follow
else:
    print('Incorrect Input')
```

When trying out pseudocode CASE solutions in Python, use the `if...elif...else` construct.

TASK

Task 3

Points for discussion:

a You know that CASE statements are more efficient than a series of IF statements. Is a sequence of `elif` statements just as efficient as CASE statements?

b How are Python's `elif` statements more flexible than CASE statements?

6.09 Drawing Flowcharts for CASE Statements

CASE statements are actually the most efficient code structure we have met for choosing different outcomes based on the value of a single variable. This is because they use the value as an index and go straight to that line of code, ignoring all others. Python's `if...elif...else` construct requires each evaluation to be made until the correct one is reached (and then the rest are skipped). This is the same situation as for nested IF statements.

The flowchart for nested IF statements and the Python solution are thus the same. The flowchart for the pseudocode in Figure 6.03(a) can be seen in Figure 6.04(a). The pseudocode in Figure 6.03(b) for the solution using CASE statements is better represented by the flowchart in Figure 6.04(b).

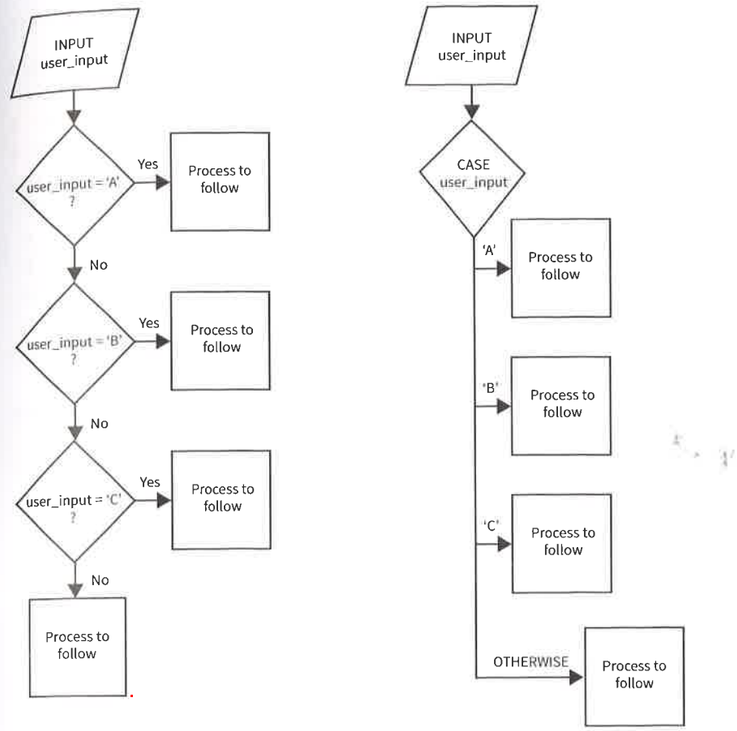


Figure 6.04 Example flowchart using nested IF statement (a) and using CASE statements (b)

6.10 Programming Tasks

DEMO TASK

Calculator (Optional)

Create a calculator that works in this way:

1. User inputs a number.
2. User selects one of four arithmetic operators.
3. User inputs second number.
4. The appropriate output is provided.

Figure 6.05 shows the flowchart for this task and Figure 6.06 shows the pseudocode option.

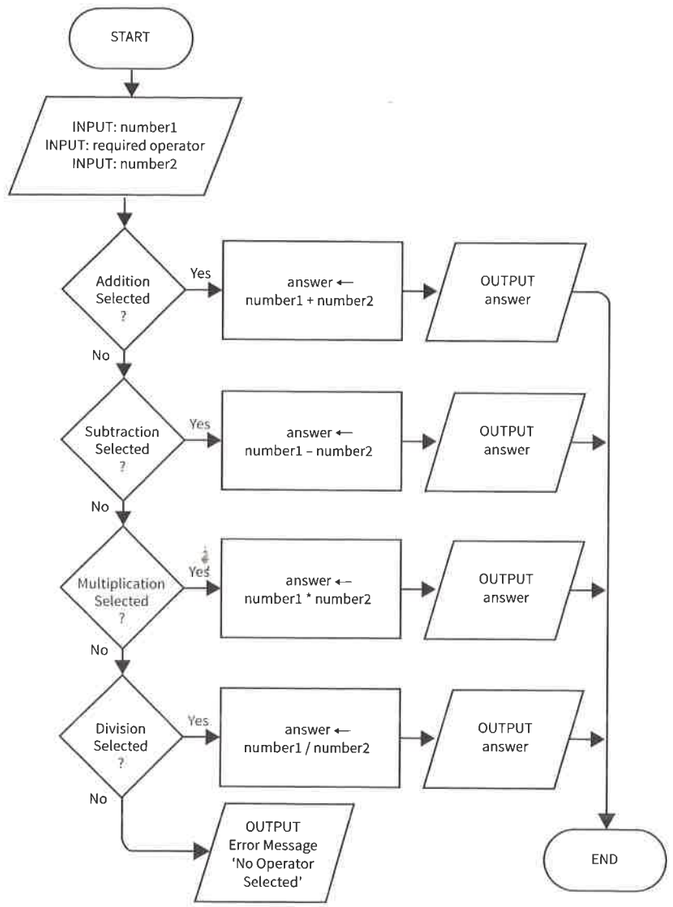


Figure 6.05 Flowchart for calculator algorithm

```
INPUT number1
INPUT operator
INPUT number2

CASE operator IF
'Add': OUTPUT number1 + number2
'Subtract': OUTPUT number1 – number2
'Multiply': OUTPUT number1 * number2
'Divide': OUTPUT number1 / number2
OTHERWISE
    OUTPUT Operator.'
ENDCASE
```

Figure 6.06 Pseudocode for calculator algorithm

The following code implements an appropriate text-based Python application:

```
number1 = int(input('Enter First Number: '))

# Use triple speech marks to enclose multiple
# line strings to maintain formatting
print("""
Choose one of the following options:
 A for Add
 S for Subtract
 M for Multiply
 D for Divide""")
operator = input()

number2 = int(input('Enter Second Number: '))

if operator == 'A':
    print(number1+number2)
elif operator == 'S':
    print(number1-number2)
elif operator == 'M':
    print(number1*number2)
elif operator == 'D':
    print(number1/number2)
else:
    print('Incorrect Operator')
```

If coded as a GUI application, the interface might look like Figure 6.07.

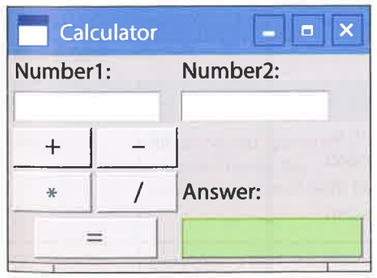


Figure 6.07 Interface for a GUI version of calculator

Producing GUI based applications is outside the syllabus.

TASK

Task 4 – Simple Calculator (Optional)

Write the code for a working calculator that has a GUI interface similar to that shown in Figure 6.07 that takes numbers with the Real data type rather than Integers.

Hints:

- `operator` will need to be a global variable.
- You will need a separate function for each operator button in your calculator. These will need to set `operator` to A, S, M or D as required.
- Real numbers are called Floats in Python.

TASKS

Task 5 – Examination Grading System

A program is required to take as input values the number of marks a student achieved and the total number of marks available in an examination. It should output the grade obtained based on the grade boundaries in Table 6.03

Table 6.03

Grade awarded	Condition
A	Student achieves 80% or more of the marks
B	Student achieves 70% or more of the marks
C	Student achieves 60% or more of the marks
U	Student achieves less than 60% of the marks

a Produce a flow diagram for a CASE solution to this problem.

b Produce the pseudocode for a CASE solution to this problem.

c Produce a text-based Python solution to this problem that uses the `if...elif...else` construct.

Task 6 – Parcel Delivery System

A system is required to calculate the delivery cost of parcels. All parcels have a fixed charge for delivery if the parcel is 5 kg or below in weight. For local deliveries this charge is £20; for international deliveries the charge raises to £40.

The maximum weight limit for international deliveries is 5 kg; however, for local deliveries extra weight is permitted and charged at £1 for every 1 kg above that limit.

a Draw a flowchart and create a pseudocode algorithm that will identify if the parcel is local or international and then apply the appropriate weight formula to calculate the correct cost.

b Test that your algorithm works by programming and running your code in Python.

Task 7 – CO_2 Calculator

A student has been asked to create a simple carbon dioxide (CO_2) calculator. The system is intended to show the difference in emissions between petrol and diesel cars. The intended inputs are the type of fuel the car uses and whether the capacity of the engine is greater than 2 litres. The user will also input the distance travelled in the car in kilometres. The emission values in tonnes of CO_2 per 1000 kilometres are shown in Table 6.04.

Table 6.04

Fuel type	Engine size 2 litres or less	Engine size greater than 2 litres
Petrol	0.208 tonnes CO_2/1000 km	0.296 tonnes CO_2/1000 km
Diesel	0.176 tonnes CO_2/1000 km	0.236 tonnes CO_2/1000 km

a Draw a flowchart and create a pseudocode algorithm that will output the tonnes of CO_2 for the distance and type of vehicle input.

b Test that your algorithms work by programming and running the code in Python.

EXTENSION TASK

Task 8 – Improved Calculator (Optional)

This task can only be implemented as a GUI application.

The calculator you made earlier requires you to input two numbers into two different input text boxes. Most calculators only have one display box for both the input and output. They follow this process:

1. Input the first number in the display box.
2. Select an operator which also clears the display box and stores the first number.
3. Input the second number in the display box.
4. Select 'equals', which performs the intended operation and displays the result.
5. Select 'clear', which clears the stored values and clears the display.

a Produce a flowchart to create a program for a calculator that has only one text box and performs the process described above.

Hint: You may need to break this up into more than one flowchart.

Hint: There may be an opportunity to extract some repeating code into a helper function.

b Produce a GUI application in Python that implements your algorithm.

Producing GUI based applications is outside the syllabus.

6.11 Connecting Logical Operators

Often a single logical operator is not sufficient to define the required criteria. For example, a fire alarm system may be required to activate if it detects either the presence of smoke or a high temperature. The logical operator in this case requires two criteria, either of which being true would cause activation of the alarm.

Python, in common with many other languages, uses the logical connectors shown in Table 6.05.

Table 6.05

Operator	Description	Example in Python
AND	**All** connected operators must be True for the condition to be met.	`if student_user == True and ID_number > 600:` The condition will only be `True` where the user is a student with an ID number higher than 600. Any other type of user with an ID number > 600 will not meet the criteria because it will fail the `student_user == True` element of the condition.
OR	**Only one** of the connected operators needs to be True for the condition to be met.	`if smoke_detected == True or temperature > 70:` The condition will be True if either smoke is detected or the temperature is above 70 °C. It will also be True if both elements are met.

NOT	Used where it is easier to define the logical criteria in a negative way.	`if not input_number == 6:`
		The condition will be True for any number input with the exception of the number 6.
		Could also have been written:
		`if input_number != 6:`

Using AND to Provide Range Criteria

In the following code, the condition is met if the input number is greater than 10 but less than 15.

```
if num > 10 and num < 15:
    # code to execute
```

Beware of getting the wrong operator. Using `or` in the statement would evaluate to true for any number.

Using the AND Operator to Replace a Nested IF Statement

A nested IF statement is often used to check two conditions.

```
if student_user == True:
    if ID_number > 600:
        # code to execute
```

When the conditions are simple, the nested IF statement can be replaced by AND. The following code represents the same conditions:

```
if student_user == True and ID_number > 600:
    # code to execute
```

Using Logical Connectors

In flowcharts, we need to be able to represent logical operators. We do this with logical connectors.

The last two code snippets are shown, represented as flowcharts in Figure 6.08.

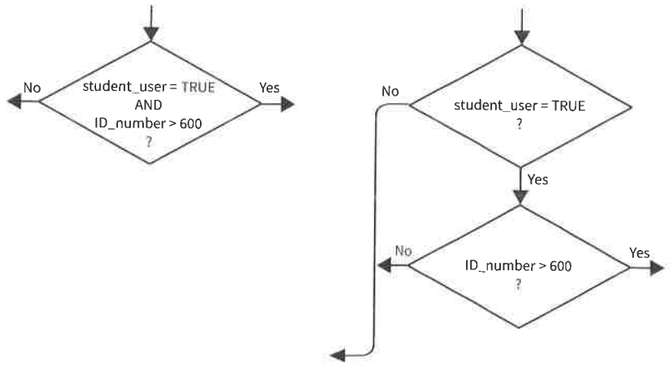


Figure 6.08 Flowchart with and without logical connectors

EXTENSION TASK

Task 9 – CO_2 Calculator Extension

If you did not already do so, rewrite the CO_2 emissions program from Task 7 using connected criteria.

The original calculator program is capable of performing only one arithmetic calculation at a time. Many calculators allow the user to enter a sequence of numbers and operators, displaying the cumulative result of the arithmetic process as the sequence is entered. An example of this process is given in Table 6.06.

Table 6.06

User input	Display	Process
43	43	
+	43	• Stores first number input • Records the addition operator selected
11	11	
*	54	• Completes the addition of the stored number with new number (43 + 11) and outputs result (54) • Records that multiplication is the latest operator selected • Stores 54 as the cumulative result
2	2	
–	108	• Completes the multiplication of the cumulative value by the new number input (54 * 2) and outputs the result (108) • Records that subtraction is the latest operator selected • Stores 108 as the cumulative result
100	100	
/	8	• Completes the subtraction of the new number from the cumulative value (108 – 100) and outputs the result (8) • Records that division is the latest operator selected • Stores 8 as the cumulative result
2	2	
=	4	• Completes the division of the cumulative value by the new number input (8 / 2) and outputs the result (4) • Records end of sequence by setting all operator indicators to False • Sets cumulative result and any interim stored numbers to 0

EXTENSION TASK

Task 10 – Cumulative Calculator (Optional)

a Produce a flowchart or pseudocode for a calculator that is capable of allowing a sequence of numbers and operators to be input. When the first operator is selected, it will store the initial value. As subsequent operators are pressed, they will complete the previous operation, storing and displaying the cumulative value. The equals button will end the sequence, display the final cumulative value and reset any variables to allow a new sequence to be input.

b Test your algorithm works by programming and running the calculator as a Python GUI application.

Producing GUI based applications is outside the syllabus.

Summary

- Selection provides methods that programmers can use to allow the algorithm to follow different paths through the code depending on the data being used at the time.
- The flowchart symbol for selection is a decision diamond.  The selection criterion is included in the symbol. The exit paths should be indicated as Yes and No or True and False.
- Multiple decisions are shown as a series of connected decisions.
- Logical operators are used to provide a range of comparative options.
- IF statements provide the ability to use complex criteria based on multiple variables or user inputs.
- A String of separate IF statements can usually be replaced by more efficient nested IFs or CASE statements.
- Nested IF statements provide the ability for additional conditions to be checked once a path has been determined by earlier conditions.
- CASE statements provide a simple method of providing multiple paths based on a single variable or user input. They are not available in Python and should be replaced with `if . . . elif . . . else`
- The ELSE statement (used with IF) and the OTHERWISE statement (used with CASE) provide a default path should none of the conditions be met.

Chapter 7: Iteration

Learning objectives

By the end of this chapter you will understand:

- the need for iteration
- how to design and represent iteration using flowcharts and pseudocode
- how to write code that will repeat instructions a predetermined number of times
- how to write code that will repeat instructions based on user input
- how to use counters with repeated code
- the advantages and disadvantages of FOR, WHILE and REPEAT UNTIL loops.

7.01 The Need for Iteration

Many processes and algorithms will complete repetitive operations on changing data. For example, a system that is required to check that a series of 100 numbers are all above a certain value would be required to check each value against the same condition. It would be impractical to reproduce that code a hundred times; a better alternative would be to rerun the same checking algorithm a hundred times with the input value changing each time the code is repeated. Another algorithm may be required to evaluate each character in a String; a loop could be used to rerun the evaluation code for each character in the String.

7.02 Types of Iteration

Three basic forms of iteration (Table 7.01) exist in the majority of programing languages. They are known as 'loops', because they cause the program to repeatedly 'loop through' the same lines of code.

Table 7.01

Loop type	Description	When it should be used
FOR loop	Repeats a section of code a predetermined number of times	The number of iterations is known or can be calculated. The programmer can set the code to loop the correct number of times.
WHILE loop	Repeats a section of code while the control condition is true	The number of iterations is not known and it may be possible that the code will never be required to run. The condition is checked before the code is executed. If the condition is false, the code in the loop will not be executed.
REPEAT UNTIL loop	Repeats a section of code until the control condition is true	The number of iterations is not known but the code in the loop must be run at least once. The condition is checked after the code has been executed, so the code will run at least once.

Often, it is possible to use any of the three types when producing an algorithm; however, each type offers the programmer certain advantages. Selecting the most appropriate type of loop can help to make your code more efficient.

7.03 FOR Loops

A **FOR loop** can only be used where the number of iterations is known at the time of programming. Often this will be in a situation when the number of iterations is 'hard-coded', but it is also possible to make use of variables when identifying the number of iterations.

SYLLABUS CHECK

Pseudocode: understand and use pseudocode, using FOR . . . TO . . . NEXT loop structures.

These are also known as 'count-controlled' loops, because the number of iterations is controlled by a loop counter. It is traditional to use a variable named i (an abbreviation for the word 'index') as the control variable.

Here is the pseudocode and Python syntax for a FOR loop:

```
FOR i 1 TO 10
  //Code to execute
NEXT
```

```
for i in range(1,11):
    # Code to execute
```

Notice how neither the pseudocode nor the Python code need to increment i within the loop; this is handled automatically by FOR loops. Do not forget, however, to add a line such as i = i + 1 to your flowcharts.

Let's look at the pseudocode first. Each individual element of the loop performs an important role in achieving the iteration as shown in Table 7.02.

Table 7.02

Element	Description
FOR	The start of the loop.
i = 1 TO 10	i is a counter variable that records the number of iterations that have been run. This is usually incremented by 1 every iteration. There is no requirement to declare the counter variable separately – it is automatically declared as part of the FOR loop. The value of the counter variable can be used within the loop to perform incremental calculations.
NEXT	The end of the iteration section. The value of the counter variable is incremented and the flow of the program goes back to the FOR line. The loop will evaluate to see if the counter value is still within the condition (10 in this example). If the counter has exceeded the end value, the loop will direct the flow of the program to the line of code following NEXT, if not it will rerun the loop.

Any code that is placed within the FOR loop will be repeated on each iteration. The repeated code can itself include complex processes such as selection or additional loops.

TIP

As the conditions are checked at FOR, NEXT will always pass execution of the loop back to FOR to check the conditions. It is a common misconception that once the maximum number of iterations has been reached, NEXT will exit the loop. This is not true. Consider a situation where a FOR loop is written to execute ten times. Although the loop counter may have reached 10, NEXT will still increment the counter to 11 before passing execution to FOR. The value of the loop counter will be outside the criteria and FOR will then exit the loop.

The Python code is somewhat different. Just as there are several types of loops in different languages, there are also different types of FOR loop. Python FOR loops are optimised for container variables (these feature in Chapter 11). There is no implementation of a counter FOR loop in Python. The FOR loop in Python instead iterates through a group of variables and finds out for itself how many items there are in that group. This is very powerful. However, to provide an implementation that is analogous to the pseudocode style FOR loop used in the syllabus, it requires the use of the `range()` function to generate a sequence.

`range()` takes three arguments:

- the Integer to start at (default = 0).
- the first Integer to exclude (required).
- the amount to increment by (default = 1).

Study the interactive sessions below to see how `range()` produces sequences of numbers. Note that the `list()` function is used to cast the sequence produced into a form that can be seen in the Python Shell. (You will learn about lists in Chapter 11.)

INTERACTIVE SESSION

Create a sequence of numbers starting at 3 and ending before 10 in increments of 2:

```
>>> list(range(3, 10, 2))
[3, 5, 7, 9]
```

To create a sequence of numbers starting at 3 and ending before 10 in increments of 1:

```
>>> list(range(3, 10)) # if left out, the default increment is 1
[3, 4, 5, 6, 7, 8, 9]
```

To create a sequence of numbers starting at 0 and ending before 10 in increments of 1:

```
>>> list(range(10)) # if left out, the default start is 0
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

So to produce a sequence of numbers from 1 to 10, we call range(1,11) and so

```
for i in range(1,11):
```

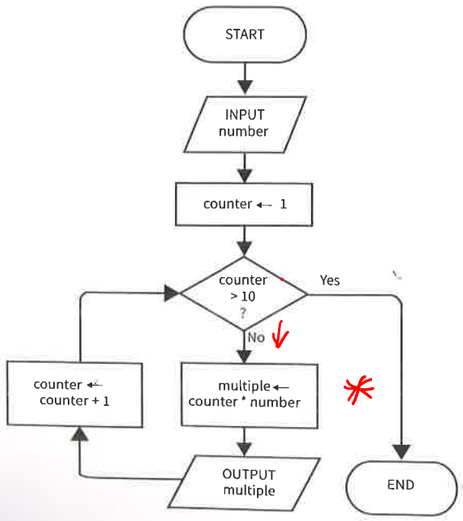
can be read as, 'For all items in the sequence [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] loop through once and refer to the current item as `i`.'

DEMO TASK

Multiplier

A system is required to output the multiples of a given number up to a maximum of 10 multiples. For example, the multiples of 6 are 6, 12, 18, 24, 30, 36, 42, 48, 54 and 60.

Figure 7.01 shows the flowchart and pseudocode for the design of the algorithm.




```
INPUT number

FOR counter 1 TO 10
    multiple ← counter * number
    OUTPUT multiple
NEXT
```

Figure 7.01 Flowchart and pseudocode for outputting multiples

SYLLABUS CHECK

Pseudocode: understand and use pseudocode for counting (e.g. Count ← Count +1).

The code for the text-based application in Figure 7.01 could be similar to the following:

```
# multiples.py

# Input the user number, cast string input to an integer
# and store in the variable number
number = int(input("Input number to multiply: "))

for counter in range(1,11):
    multiple = number * counter
    print(multiple)
```

If the user enters 4, then the output from this program would be:

```
4
8
12
16
20
24
28
32
36
40
```

The GUI application in Figure 7.02 makes use of a text box that holds multiple lines of text. The layout has been achieved without using frames. The window size and colour have been provided by using tkinter's `geometry()` and `configure()` methods. A trick has been used to provide vertical spacing above the label. Can you see how this was achieved?

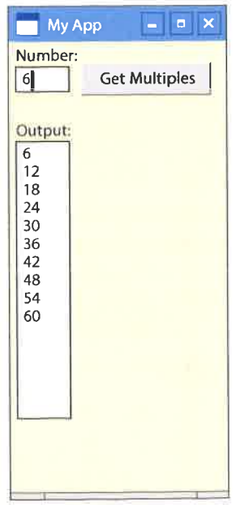


Figure 7.02 GUI application of multiplier

Producing GUI based applications is outside the syllabus.

Items are added to the list box by using the code:

```
<Text box name>.insert(END, <string to add>)
```

The data to be added must be a String but as the result of the calculation is an Integer, it has to be cast with `str()` before it can be added to the text box. The code to achieve Figure 7.02 would be:

```
# multiples-gui.py

from tkinter import *

def multiply():
    # get contents of textbox_input
    number = int(textbox_input.get())

    # clear output text box
    textbox_output.delete(0.0, END)

    # process and output result
    for counter in range(1,11):
        multiple = str(number * counter) + '\n'
        textbox_output.insert(END, multiple)

# Build the GUI
window = Tk()
window.title('My App')

# give the window a size and background colour
window.geometry('150x350')
window.configure(background='linen')

# Create the labels
input_label = Label(window, text='Number: ', bg='linen')
input_label.grid(row=0, column=0)
output_label = Label(window, text='\nOutput: ', bg='linen')
output_label.grid(row=2, column=0)

# Create text entry box for entering number
textbox_input = Entry(window, width=5)
textbox_input.grid(row=1, column=0)

# Create text box for outputting multiples
textbox_output = Text(window, height=15, width=6)
textbox_output.grid(row=3, column=0)

# Create the button
multiply_button = Button(window, text='Multiply', command=multiply)
multiply_button.grid(row=1, column=1)

window.mainloop()
```

The trick to provide the vertical space above the label was to include a line return `\n` before the rest of the label text.

TASKS

Task 1

Extend the multiply system to include two inputs. The first input is the number to multiply, the second is the number of multiples required.

Task 2

Produce a system that accepts two numbers, a and b, and outputs a^b. For example, if a = 3 and b = 4, the output will be 81 ($3^4 = 3 \times 3 \times 3 \times 3$).

7.04 Using Loops with Advanced Arithmetic Operators

In Chapter 3, we learnt about the use of basic arithmetic operators such as add and divide. Python offers more advanced arithmetic operators.

A particular example of this is the way in which we divide numbers. It would be normal to expect the result of a division to be an exact value such as 91/24 = 3.792. But if you are dealing with data that can only be represented as Integers then a different approach might be needed.

If you had to organise transport for a group of 91 people using buses that can hold a maximum of 24 people you are more likely to want to express 91/24 in the format '3 remainder 19'. This format would allow you to identify that you would need four buses and still have seats available for another five people. The arithmetic terms for these items are 'quotient' and 'modulus' (Table 7.03) and they prove useful when producing certain mathematical algorithms.

Hopefully you'll remember these

Table 7.03

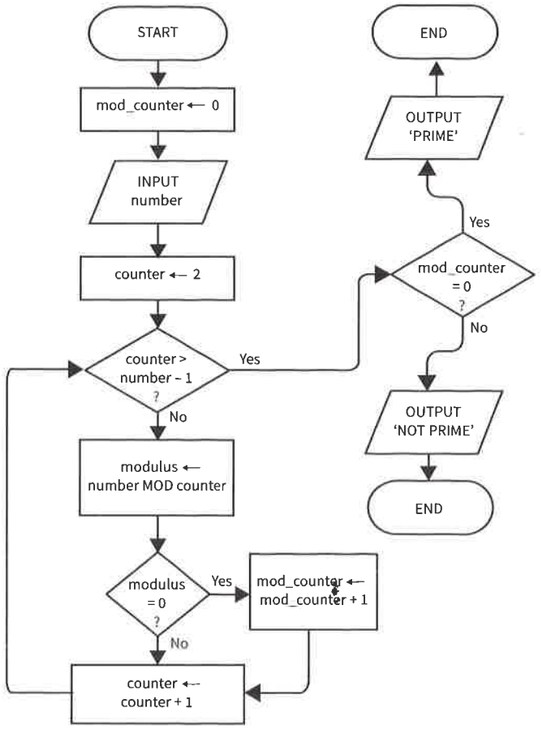
Operator	Description	Python code
Quotient	A division operation that outputs the Integer part of the result. This is also known as Integer division.	The Integer division operator is two slashes: `91//24 = 3`
Modulus	A division operation that outputs the remainder part of the result. The amount by which one number will not exactly divide into another.	The modulus operator is a percentage symbol: `91%24 = 19`

Figure 7.03 shows the flowchart and pseudocode for a solution.

DEMO TASK

Prime 1

A prime number can only be divided equally by 1 or itself. If a number can divide equally into another number, the modulus of that operation will be zero. Therefore a prime number can also be defined as a number that, when divided by all the positive Integers between 1 and itself, will not result in a modulus of zero. A system is required that uses this rule to determine if a number is a prime.




```
mod_counter ← 0

INPUT number

FOR counter 2 TO number - 1
   modulus ← number MOD counter
   IF modulus = 0 THEN
      mod_counter ← mod_counter + 1
   END IF
NEXT

IF mod_counter = 0 THEN
   OUTPUT 'Prime Number'
ELSE
   OUTPUT 'Not a Prime'
END IF
```

Figure 7.03 Flowchart and pseudocode for the prime algorithm

Note that in both designs, the iteration `counter` starts from 2 and ends at `number-1` to avoid the use of 1 and the number that is input occurring in the loop. These would result in a modulus of zero. To achieve the same thing when using the `range()` function in Python, we provide the arguments 2 and `number`.

TASK

Task 3 – Discussion Question

Is limiting the iterations to `number-1` the most efficient range limiter? What might be a better option?

Can the program work more efficiently

The following is the code for a text-based Python implementation:

```
modulus_counter = 0

number = int(input('Enter your number: '))

for counter in range(2,number):
    modulus = number % counter
    if modulus == 0:
        modulus_counter = modulus_counter + 1

if modulus_counter == 0:
    print('Prime number.')
else:
    print('Not a prime number.')
```

TASK

Task 4 (Optional)

Design a GUI version of this application giving a background colour of your choice to the window.

Producing GUI based applications is outside the syllabus.

7.05 Condition-controlled Loops

The WHILE . . . DO . . . ENDWHILE and REPEAT . . . UNTIL loop structures are controlled by a specific condition. Iterations are repeated continuously based on certain criteria. These types of loops allow iteration where the number of repetitions is unknown.

Consider the situation where one random number was constantly subtracted from another until the resultant value was less than zero. It would be impossible to determine the number of iterations required to cause the first number to be less than zero. As a result, a FOR loop would not be appropriate; another iterative method would have to be used.

While Loops

Iterations continue while the loop conditions remain true irrespective of the number of iterations this may generate. It is usual for the code within the loop to have an impact on the conditional values of the loop in such a way that the criteria will eventually become false and the loop will cease.

Because the conditions are tested at the start, it is possible that the loop will never run if the conditions are false at the outset. It is also possible to inadvertently code an infinite loop where the conditions remain true for ever.

When writing a **WHILE loop** in Python you use the following format:

```
while counter > 0:
    # code to be iterated
    counter = counter - 1
```

This is very similar to the Cambridge IGCSE and O Level Computer Science pseudocode format:

```
WHILE counter > 0 DO
   //code to be iterated
   counter = counter – 1
ENDWHILE
```

KEY TERMS

WHILE loop: A type of iteration that will repeat a sequence of code while a set of criteria continue to be met. If the criteria are not met at the outset the loop will not run and the code within it will not be executed.

SYLLABUS CHECK

Pseudocode: understand and use pseudocode, using WHILE . . . DO . . . ENDWHILE loop structures.

Each individual element of the loop performs an important role in achieving the iteration, as shown in Table 7.04.

Table 7.04

Element	Description
`while`	The start of the loop
`counter > 0`	The condition that controls the loop. Each time the iteration is run, the condition is evaluated and if it remains True, the iteration will run. Once the condition is False, execution of the code is directed to the line following the loop. In counter-controlled WHILE loops, it is important that code is included within the loop to increment or decrement the counter. In a FOR loop, the counter is automatically incremented. The same facility does not apply to WHILE loops and, as a result, the programmer must include appropriate code.
end of indented code	The end of the current iteration. Execution of the program returns to `while` so the condition can be re-evaluated and further iterations actioned. Do not forget to add ENDWHILE when writing pseudocode.

TIP

Remember that WHILE loops iterate while the condition evaluates to True. It is possible to create an infinite loop rather easily:

```
>>> while True:
    print('Hello', end='')
```

It is therefore important to know how to break out of infinite loops. To do so, hold down the *CTRL* key on your keyboard and press *C*. Try the code above yourself in an interactive session. The optional parameter `end=''` provided in the `print()` function suppresses the default line return.

In the multiplier demo task, a system was required to output the multiples of a given number up to a maximum of ten multiples. This can also be coded with a WHILE loop.

CTRL-C breaks out of an infinite loop

Compare the flowchart and pseudocode in Figure 7.04 with the FOR loop example in Figure 7.01.

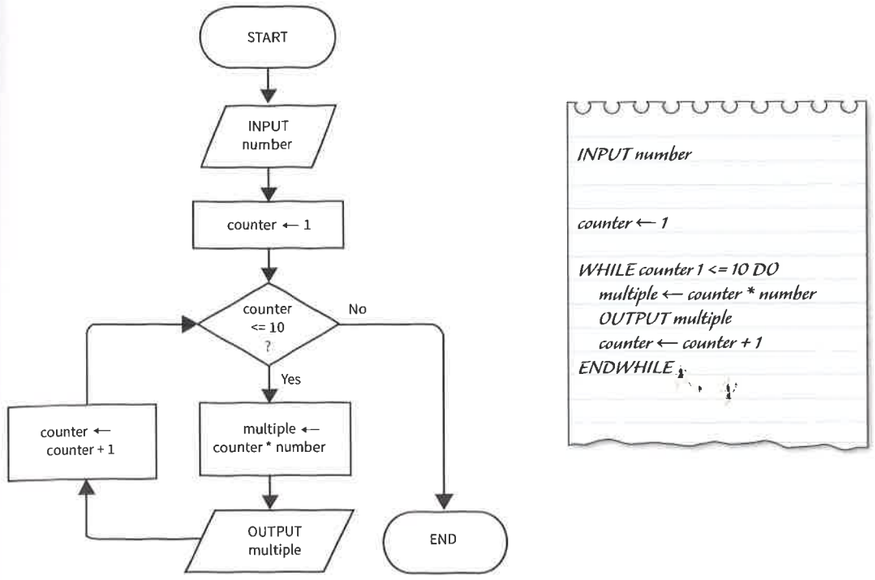


Figure 7.04 Flowchart and pseudocode for a WHILE loop

The flowcharts are nearly identical. A FOR loop always evaluates to see if the counter has reached its target and, if it has not, keeps looping. A WHILE loop can evaluate any condition and keeps looping while the condition is True. Thus, when we add a counter and evaluate against that counter, it should not be a surprise that the flowcharts are similar. The only difference is that the logical criteria are reversed in the flowchart.

The format of the WHILE loop has been followed in the pseudocode and the iterated code includes a line which increments the counter.

The following is the code for a text-based implementation:

```
number = int(input('Input number to multiply: '))

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

CTRL-C

TASKS

Task 5

After reminding yourself how to break out of an infinite loop (see the previous Tip), try running the code above but first comment out the last line (`counter = counter + 1`). Explain the outcome of this change.

Task 6 – WHILE Loop

A WHILE loop could be used to calculate the quotient and modulus without using the built-in operators (// and %). The WHILE loop would be set to continually subtract number b from number a while a remains greater than or equal to b. When this condition is no longer true:

- the number of subtractions is equal to a quotient b
- the number remaining is equal to a modulus b.

1 Draw a flowchart and create a pseudocode algorithm that takes as input two numbers and outputs the quotient and modulus resulting from the division of the two numbers.

2 Test that your algorithm works by programming and running the code as a Python text based application.

WHILE Loops with Multiple Criteria

In the Demo Task Prime 1 in Section 7.04, the output was a message indicating if the input number was a prime or not. To achieve this, the FOR loop may have iterated many times even though the non-prime nature of the number had already been determined.

For example, any even number can be shown not to be prime just by taking the modulus division of 2, making all subsequent iterations unnecessary. A WHILE loop can provide a more efficient solution to the prime number task by looping only while no positive divisor has been tested. The loop can end at the identification of the first positive divisor or when all relevant Integer values have been tested.

SYLLABUS CHECK

Problem-solving and design: comment on the effectiveness of a given solution.

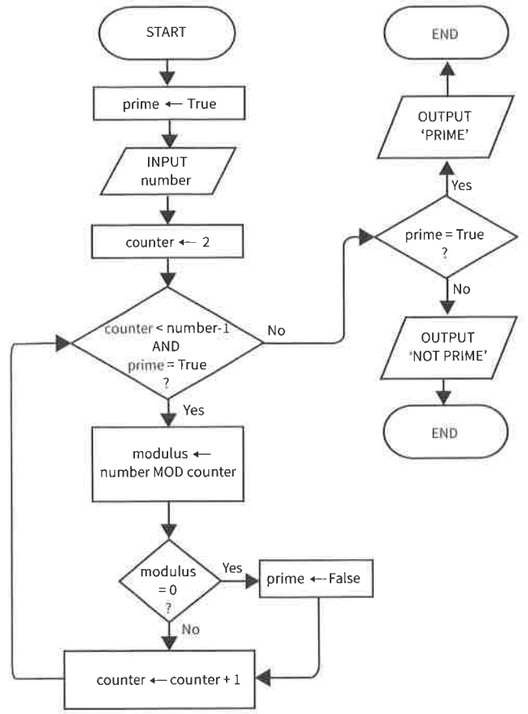
DEMO TASK

Prime 2

Construct a flowchart and pseudocode to create a WHILE loop that identifies whether a given number is a prime number or not. Then provide a Python implementation to test your solution.

The WHILE loop will need multiple criteria. In the previous algorithm a count of the number of exact divisors was maintained. In this example a Boolean value is used to indicate that an exact divisor has been identified.

Figure 7.05 shows the flowchart and pseudocode for a solution.




```
prime ← True

INPUT num

counter ← 2
WHILE counter < num-1 AND prime = True DO
   modulus ← num MOD counter
   IF modulus = 0 THEN
      prime = False
   END IF
   counter ← counter + 1
ENDWHILE

IF prime = True THEN
   OUTPUT 'Prime Number'
ELSE
   OUTPUT 'Not a Prime'
END IF
```

Figure 7.05 Flowchart and pseudocode for WHILE approach

The following is the code for a text-based implementation:

```
prime = True

number = int(input('Input number to test: '))

# declare counter to use with the while loop
# initialise to 2 to avoid using 1 as a divisor
counter = 2
while counter < number-1 and prime == True:
    modulus = number % counter
    if modulus == 0:
        prime = False
    counter = counter + 1

if prime == True:
    print('Your number is PRIME')
else:
    print('Your number is NOT PRIME')
```

REPEAT . . . UNTIL Loops

A **REPEAT . . . UNTIL loop** is very similar to a WHILE loop as iteration will continue based on the loop conditions. It is therefore also able to work in situations where the number of iterations is unknown.

KEY TERMS

REPEAT . . . UNTIL loop: A type of iteration that will repeat a sequence of code until a certain condition is met.

Unlike in a WHILE loop, the test is completed at the end of the iteration so the iterated code will always run at least once.

SYLLABUS CHECK

Pseudocode: understand and use pseudocode, using REPEAT . . . UNTIL loop structures.

There is no REPEAT . . . UNTIL loop in Python, but your pseudocode can be implemented in Python by creating an infinite loop and then using `break` after testing at the end of the loop:

```
while True:
    # Code for iteration
    counter = counter + 1
    if counter > 10:
        break
```

The individual elements of the code perform an important role in the iteration as shown in Table 7.05.

Table 7.05

Element	Description
`while True`	The start of the loop. By replacing the normal condition that is required to evaluate to `True` with `True`, we ensure that this loop will continue forever. At every iteration, the execution of the program will be passed to the `while True` command. Because the loop starts before any conditions are checked, the iteration will always run at least once.
`break`	Stop looping.
`counter > 10`	The condition for the loop. Each time the iteration is run, the condition is evaluated. If it remains `False`, the execution is directed to `while True` and the iteration will run again. Once the condition is `True`, the execution of the code is directed to whatever code follows the loop. It is possible to use the logical operators (`and`, `or` and `not`) to structure multiple conditions. Counter-based conditions require the counter to incremented by the code contained within the loop.

TIP

Because the check is made at the end of the sequence of code, the loop will always run at least once.

DEMO TASK

Prime 3

Formulate an algorithm to calculate if a number is prime using a REPEAT . . . UNTIL loop.

The flowchart and pseudocode for the REPEAT . . . UNTIL loop are shown in Figure 7.06. If you compare these with the WHILE approach in Figure 7.05, you will be able to identify the differences in approach. The decision criteria are checked at different stages during the process, the WHILE at the outset of the loop and the REPEAT . . . UNTIL at the end. The loop decision for the WHILE is based on the criteria being True and the REPEAT . . . UNTIL loops if the criteria are False.

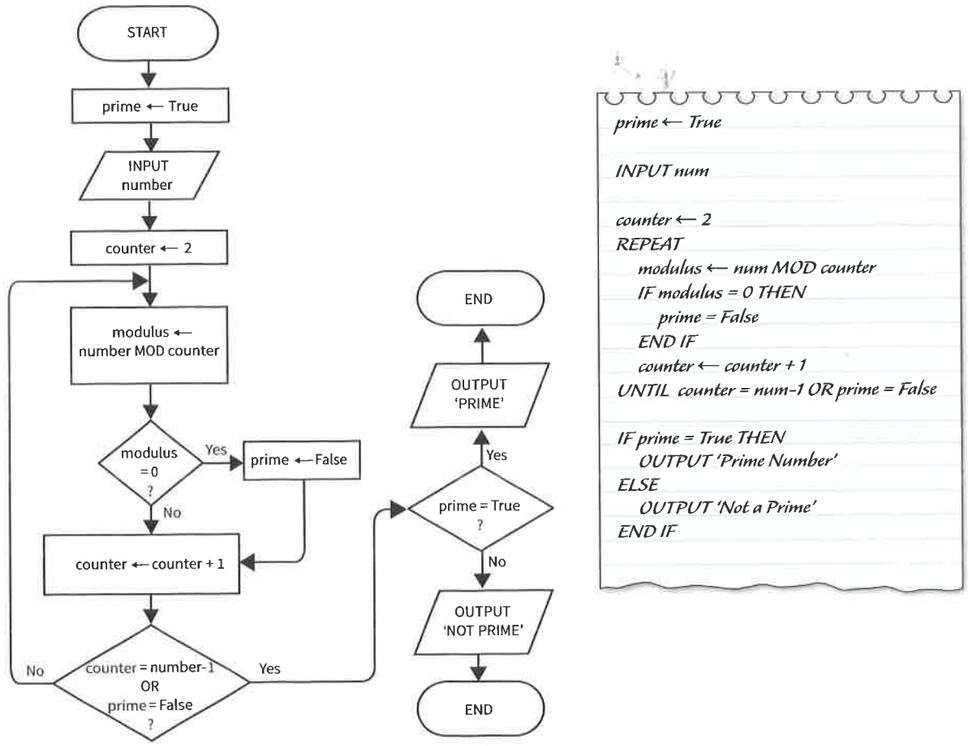


Figure 7.06 Flowchart and pseudocode for REPEAT . . . UNTIL approach

TASK

Task 7

Using the flowchart and pseudocode in Figure 7.06, produce a text-based Python implementation.

WHILE and UNTIL Criteria

When considering the criteria for a WHILE or REPEAT . . . UNTIL loop, it is important to remember that the logic for each loop is defined as the opposite of the other. For example, if the criteria was based on a Boolean value, the condition would be as shown in Table 7.06.

Table 7.06

Criteria	When Boolean = False	When Boolean = True
WHILE Boolean = False DO ENDWHILE	Continues to iterate because the Boolean is False.	Ends iteration because the Boolean is not False.
REPEAT UNTIL Boolean = True	Continues to iterate because the Boolean is not True.	Ends iteration because the Boolean is True.

7.06 WHILE and REPEAT . . . UNTIL Loops Based on User Input

As both WHILE and REPEAT . . . UNTIL loops are capable of iterating an unknown number of times, they are able to work in an environment where a user will input a sequence of data items and indicate the end of the sequence by inputting a specific item. Often, the data items will be a series of positive numbers and the input that will end the series is a negative number.

For example, a student is completing an experiment in which they record the height and gender of all the students who they meet in a certain time period. They are asked to indicate the end of the input process by entering a negative value for height. The program will record the average height for each gender and the total number of records entered.

In this scenario, the input would be placed within either a WHILE or a REPEAT . . . UNTIL loop with the loop condition being based on the height input value. The algorithm to maintain the total number of records entered and the height averages would iterate in the loop. The output of the system would most likely be programmed to display only when the loop had ended.

In a GUI event-driven application, it is difficult to recreate this type of scenario because of its event-driven nature which makes the inclusion of input within a loop impossible. However, text-based programs can be used to show how these types of program can be written and executed.

When would you use a while loop?
To Indicate the end of a process

TIP

When using iteration based on user input, it is crucial that the input is included within the loop. A common error is to include a single input outside the loop:

```
INPUT number
WHILE number > 6 DO
    OUTPUT number
ENDWHILE
```

Consider the situation if the user input the number 10. The loop will continually check against the value of 10 and run an infinite number of times. Again, an infinite loop is produced.

SYLLABUS CHECK

Pseudocode: understand and use pseudocode for totalling (e.g. Sum ← Sum + Number).

DEMO TASK

Summing User Input

A system is required which will allow a user to input a series of positive numbers indicating the end of the sequence by inputting a value of –1. The system will output the sum of the positive numbers input.

The flowchart in Figure 7.07 does not indicate either a WHILE or REPEAT . . . UNTIL loop – it simply includes the criteria in the loop.

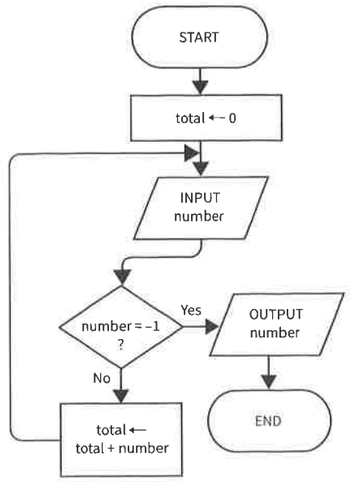


Figure 7.07 Flowchart for input loop

The different approaches can be seen in the pseudocode.

The WHILE loop in Figure 7.08(a) requires the first number to be input outside the loop to provide a value to check.

The REPEAT . . . UNTIL loop in Figure 7.08(b) will have included the input of –1 in `total` as the conditions are not checked until after the processing in the loop has been completed. Consequently, the total has to be recalculated after the loop.

(a)

```
total ← 0
INPUT number
WHILE number is not -1 DO
    total ← total + number
    INPUT number
ENDWHILE
OUTPUT total
```

(b)

```
total ← 0
REPEAT
    INPUT number
    total ← total + number
UNTIL number = -1
total ← total + 1
OUTPUT total
```

Figure 7.08 Pseudocode for WHILE and REPEAT . . . UNTIL

TASK

Task 8

a Using the pseudocode in Figure 7.08(a), produce a text-based Python WHILE loop implementation.

b Using the pseudocode in Figure 7.08(b), produce a text-based Python REPEAT . . . UNTIL loop implementation.

EXTENSION TASKS

Task 9

The algorithms in this chapter for identifying if a number is a prime are far from perfect. It is possible to come up with far superior algorithms. You could have a competition with your fellow students to see who can produce the fastest algorithm by adding a timer to your code.

To do this you need to import Python's `time` module, get a timestamp after the user has supplied input and then, after displaying the result, collect another timestamp and use this to calculate the elapsed time.

The WHILE loop program is shown below with the timing code added:

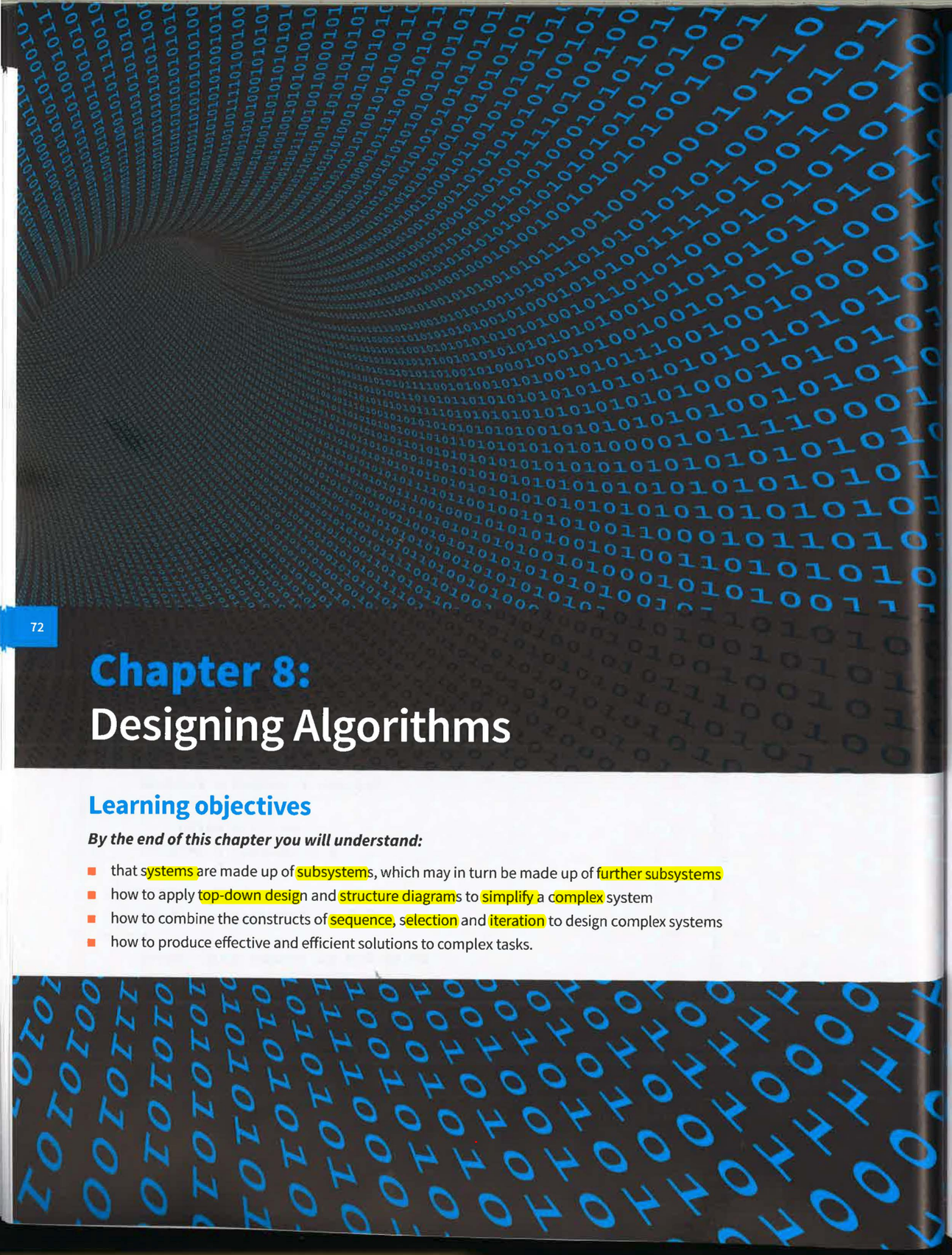
```
# Import time module to allow testing of efficiency
from time import *

prime = True

number = int(input('Input number to test: '))

# Grab start time
start_time=time()

counter = 2
while counter < number and prime == True:
    modulus = number % counter
    if modulus == 0:
        prime = False
    counter = counter + 1

if prime == True:
    print('Your number is PRIME')
else:
    print('Your number is NOT PRIME')

# Print out current time minus start_time
print('This took', (time()-start_time), 'seconds')
```

To be fair, you will need to ensure you supply a large prime number so that your algorithms are forced to go through all their tests. You will also need to test your algorithms on the same computer.

(982451653 is a suitable large prime and 982451649 is not.)

Summary

- Iteration provides methods that programmers can use to loop through sequences of code multiple times.
- There are three main types of iteration: FOR . . . TO . . . NEXT, WHILE . . . DO . . . ENDWHILE and REPEAT . . . UNTIL loops.
- Flowcharts symbolise loops through the use of a decision with a flow line looping back to an earlier element of the diagram. The decision contains the criteria on which the iteration is based.
- A FOR . . . TO . . . NEXT loop is used where the number of iterations is known at the outset.
- Condition-controlled loop structures, such as WHILE . . . DO . . . ENDWHILE or REPEAT . . . UNTIL, are used where the number of iterations is unknown.
- WHILE . . . DO . . . ENDWHILE structures check the loop conditions at the outset of the loop. If the conditions are False, the loop will never run.
- REPEAT . . . UNTIL structures check the loop conditions at the end of the first iteration. Consequently, the loop will always run at least once.
- WHILE . . . DO . . . ENDWHILE loops continue to iterate while the condition equates to True.
- REPEAT . . . UNTIL loops continue iterating while the condition equates to False.
- To implement a counting FOR loop in Python, the `range()` function is used.



Chapter 8: Designing Algorithms

Learning objectives

By the end of this chapter you will understand:

- that systems are made up of subsystems, which may in turn be made up of further subsystems
- how to apply top-down design and structure diagrams to simplify a complex system
- how to combine the constructs of sequence, selection and iteration to design complex systems
- how to produce effective and efficient solutions to complex tasks.

8.01 The Approach

When designing algorithms, you will be making use of computational thinking. This requires the ability to analyse a scenario-based task, identify the individual elements of the task and use programming concepts to create an appropriate algorithm. Often more than one approach to a given scenario would produce a working solution, which makes this process exciting. Identifying and designing an efficient solution to a problem is at the heart of computational thinking.

For your paper 2 exam you will be given a scenario-based task (Pre release task)

SYLLABUS CHECK

Problem solving and design: use top-down design and structure diagrams.

8.02 Top-down Design

Top-down design is an approach to structured programming where the problem is defined in simple steps or tasks. Each of these tasks may be split into a number of smaller subtasks. The process is complete once the problem has been broken down sufficiently to allow it to be understood and programmed. This process is also known as 'step-wise refinement'.

The aim of this technique is to break down the main task into smaller tasks

The main advantage of designing in this way is that the final process will be well structured and easier to understand. It can increase the speed of development as different subtasks can be given to individual members of a programming team. This design approach also helps when debugging or modifying as changes can be made to individual subtasks without necessarily having to change the overall program.

advantages
well structured
easier to understand
easier to fix-debug

This approach is effective in solving large, real-world problems as well as the type of scenario-based questions you may meet during an examination.

FURTHER INFORMATION

Top-down design is not the only way of solving large real-world problems. Object-oriented programming also involves breaking up large problems into smaller packages, but uses a bottom-up approach. This syllabus focusses on top-down design.

KEY TERMS

Top-down design: a method of simplifying the main system into its subsystems until the whole is sufficiently defined to allow it to be understood and programmed.

Structure diagrams: a diagrammatical method of expressing a system as a series of subsystems.

8.03 Structure Diagrams

One tool available to programmers when trying to design a solution to a more complex system is the Structure Diagram. Using top-down design principles, the aim is to decompose the system into smaller and smaller problems until no further decomposition can take place. At this point the programmer starts to develop algorithms for each subproblem in the usual way, using flowcharts or pseudocode.

Top down design-technique
structure diagram-tool

DEMO TASK

A school requires a system that will check if students are present in lessons and report to parents any absence. The teacher will record if students are in the lesson and then transfer the record to the school administration team. They will contact parents if students are absent. The process of reporting to parents can be done either by telephone or email.

Scenario task

One possible strategy is to consider the situation in terms of the steps of a computer model:

Input → Storage → Process → Output.

When breaking down the problem identify parts to the system in relation to INPUT – STORAGE → PROCESS → OUTPUT

We could start the design by considering the main elements of the required system:

- The main input will be the record of the students' presence.
- To store this some form of list could be used.
- The process will involve the following sequence of steps:

1. recording the presence of every student who attends the lesson (this will be an iterative process).
2. passing the list to the administration team who will contact parents.

Figure 8.01 shows the initial structure diagram for this system.

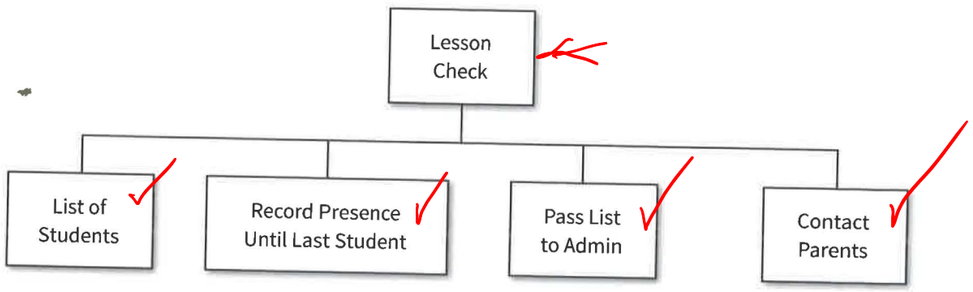
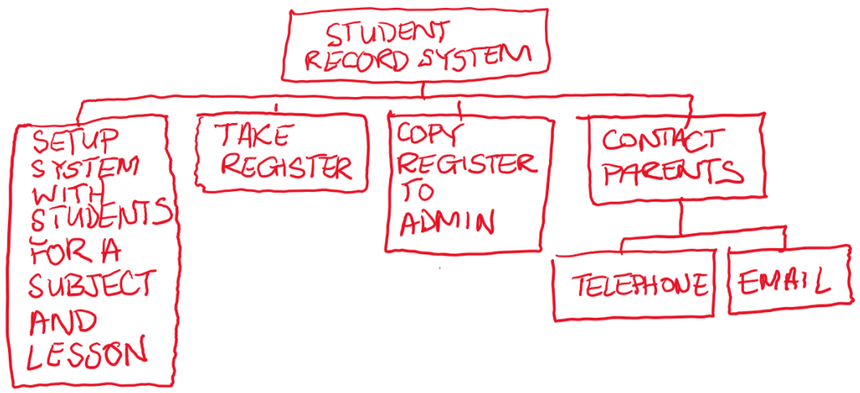


Figure 8.01 Initial structure diagram



The next step is to consider if any of the tasks could be broken down into subtasks. Creating the 'List of Students' is a high-level task that could be broken down into subtasks. Contacting parents can be done by telephone or email and will therefore need subtasks. Figure 8.02 shows the amended structure diagram with these subtasks.

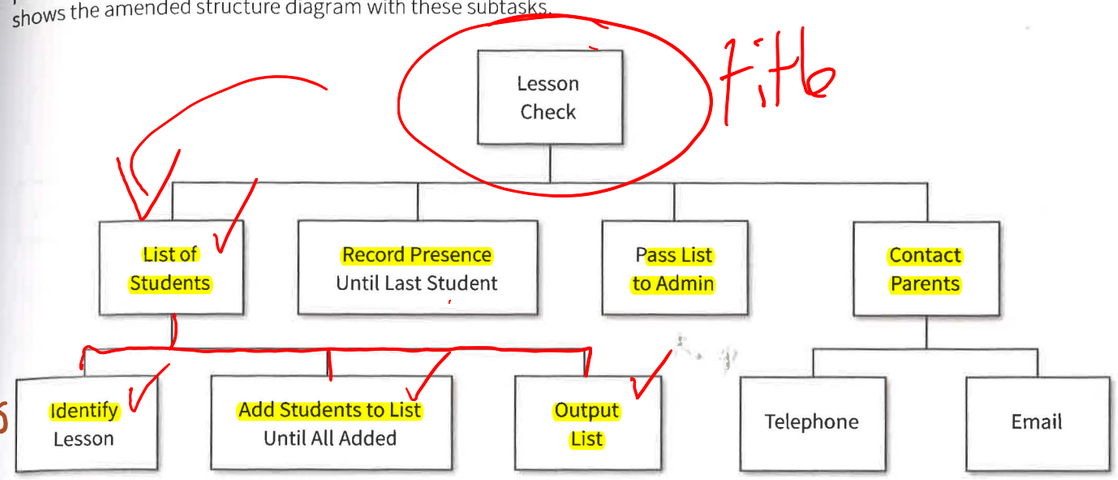


Figure 8.02 Structure diagram showing subtasks

The diagram may not yet be complete. The process 'Identify Lesson' could be broken into further subtasks. What inputs would be needed? Where is the data about which students attend the lesson stored? How will the system access those records? This and other tasks would require consideration if this was a real-life scenario.

8.04 Design Steps

As suggested above, splitting the overall task into subtasks can be done following the Input → Storage → Process → Output computer model. This will be required when preparing the pre-release material for Paper 2 (see Chapter 12). When designing an algorithm for a subtask, it is recommended that a slight adaption is made to the process:

1. Identify the inputs and outputs that are involved in the scenario. At this stage, it is worth identifying any global variables that will be required.
2. For each input, identify if the task requires this input to be repeated. This will mean some form of iteration is needed. Identify the most appropriate loop to use.
3. For each output, identify the required calculation or recording process required to produce the output value.
   a. Does the process involve any decision making? This could mean use of an IF statement.
   b. Does the process involve repeated calculation? This could mean some form of iteration.
4. Consider the sequence in which the various processes need to be completed:
   - Check that inputs or processes that need to be iterated are within the loop.
   - Check that single inputs and outputs are outside the loop.
   - Check any iteration repeats as expected.
   - Check you have defined and initialised the variables or constants that are to be used.

Considering the inputs and outputs at the start will help you to consider the aim of the system. If you don't consider the required outputs early in the design stage, how can you define the process required to produce the outputs?

Although considered at the outset, variables can be finalised as a last step to making sure nothing has been missed.

TIP

When programming, the IDE will alert you to incorrect statements and missing variables. When designing in pseudocode, this support is not available. Always check that you have initialised all global variables correctly and that program statements are complete.

SYLLABUS CHECK

Problem solving and design: produce an algorithm for a given scenario in the form of a flowchart or pseudocode.

DEMO TASK

Design: Known Quantity of Inputs

A user will input 100 positive numbers into a system. The system will output the highest number and the sum of the numbers input. Design an appropriate algorithm.

Figure 8.03 outlines the computational thinking process for designing the algorithm using the steps described above.

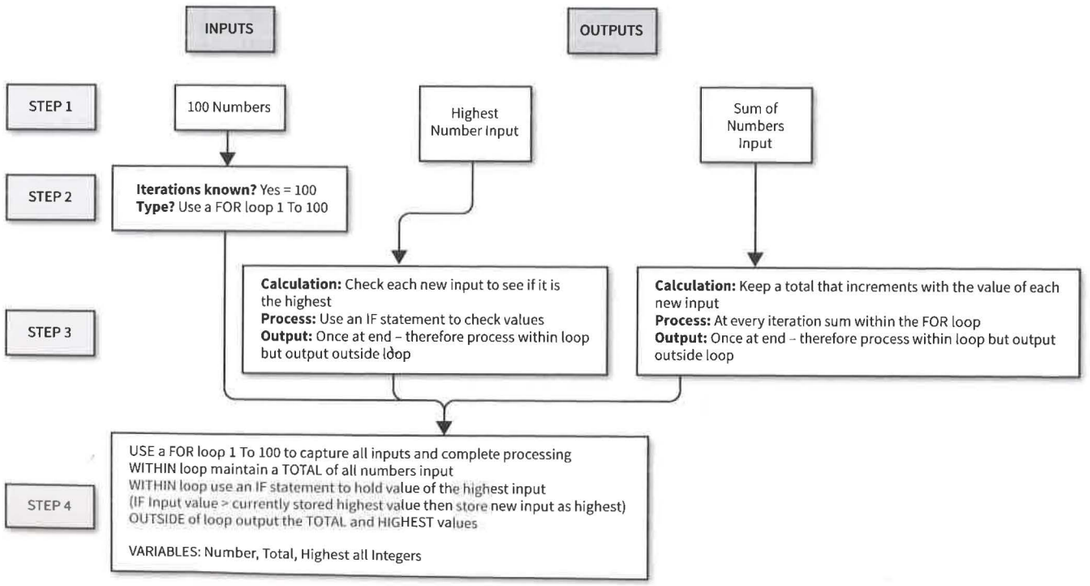


Figure 8.03 The steps in computational thinking

The structure diagram in Figure 8.04 shows the design of the process.

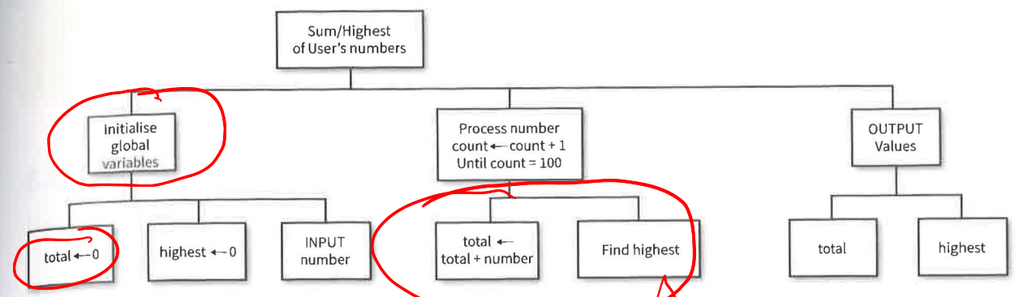
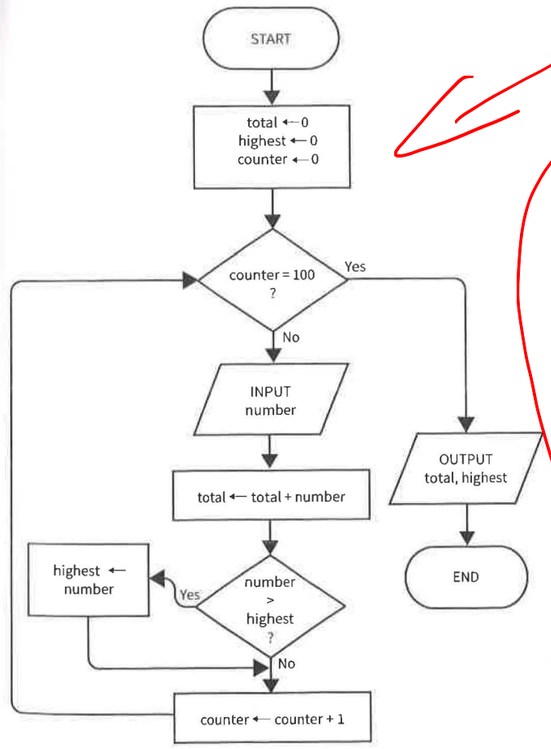
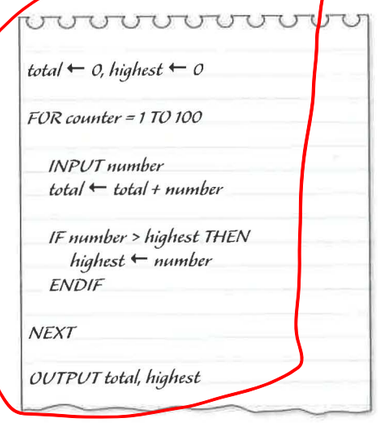


Figure 8.04 Structure diagram

The resultant flowchart and pseudocode are shown in Figure 8.05.





```
total ← 0, highest ← 0
FOR counter = 1 TO 100
    INPUT number
    total ← total + number
    IF number > highest THEN
        highest ← number
    ENDIF
NEXT
OUTPUT total, highest
```

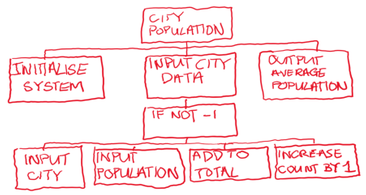
```
total = 0;highest = 0
for counter in range (1, 11):
    number = int(input('Number: '))
    total = total + number
    if number > highest:
        highest = number
print('Total is: ',total,' Highest is: ',highest)
```

Figure 8.05 Flowchart and pseudocode approach

TASKS

Task 1 – Discussion Question

This is not the only acceptable solution. Identify two other ways this algorithm could have been written.



Task 2 – Design: Unknown Quantity of Inputs

A user is required to input the population of a number of local cities. They will indicate the end of the input sequence by inputting a negative value. The system will output the average population of all the cities.

Total ← 0
Count ← 0
INPUT City
WHILE City <> -1
  DO
    INPUT Population
    Total ← Total + Population
    Count ← Count + 1
    INPUT City
ENDWHILE
IF Total <> 0:
  OUTPUT "Total is: "
  OUTPUT Total
  OUTPUT "Average population is: "
  OUTPUT Total/Count
ENDIF

TIP

To check that your solution works as intended it should be tested. See Chapter 10 to discover how this can be achieved.

8.05 The complete design process

When presented with a complex problem, any or all of the following processes might be needed to create a finished solution:

- Decomposition – using **top-down design** and **structured diagrams**
- Standard methods of solution – using **flow charts** and **pseudocode** initially and then examining the algorithm to see if it could be **simplified**, or made more **efficient** by employing loops or subroutines
- Check input data – This is called **validating** user input (see Chapter 9)
- Testing – thorough **testing** needs to then take place (see Chapter 10)

```
total = 0
count = 0
city = input('Input city or -1 to quit: ')
while city != str(-1):
    population = input('Population: ')
    total = total + int(population)
    count = count + 1
    city = input('Input city or -1 to quit: ')
if total != 0:
    print('Total is: ', total)
    print('Average population is: ', total/count)
```

Simplifying algorithms makes for more reliable and maintainable solutions. Efficient algorithms use loops and functions to collect repetitive input and processing. An efficient algorithm will combine the collection of multiple processes in one loop rather than in separate loops wherever possible.

You are expected to be able to design effective solutions and comment on the effectiveness of algorithms presented to you. Remember to check whether the solution could be simplified further or made more efficient as well as whether it handles all your test data (Chapter 10), and that all input is effectively validated (Chapter 9). The efficiency of different solutions is examined in more depth in Chapter 7 where the use of different kinds of loops is compared, and again in Chapter 12 where advice is given on how to prepare for the pre-release tasks for Paper 2. Preparing for the pre-release tasks will involve most, if not all, of the processes above.

SYLLABUS CHECK

Explain standard methods of solution.
Comment on the effectiveness of a given solution.

KEY TERMS

Standard methods of solution: The two main standard methods of solution used throughout this book are flowcharts and pseudocode. Other standard methods of solution include, top-down design, structure diagrams, simplification, efficiency analysis, validation and testing.

8.06 Design Challenges

The following is a series of examination-style tasks.

Include a structure diagram

For each task, design an appropriate algorithm using a flowchart and pseudocode. Examples of working algorithms are included in Chapter 14, but remember these are not the only possible solutions.

TASKS

Task 3

A student is completing a mathematical probability study. They are required to throw a standard six-sided dice 100 times. They will input the number shown at each throw into a system. The system will output the number of times the dice shows a six and the average value of all the throws.

Task 4

The student completing the mathematical probability study in Task 3 decides to extend the study to include two standard six-sided dice. The student will throw both dice simultaneously, inputting both numbers shown at each throw into a system. The system will record the number of 'doubles' (throws that result in both dice showing the same number). The process will end when 100 doubles have been recorded, at which time the system will output the percentage of total throws that resulted in a double.

complete for H/w.

Task 5

A user inputs into a system a sequence of positive numbers. They indicate the end of the sequence by inputting the value -1. The system will output the highest and lowest numbers input.

Task 6

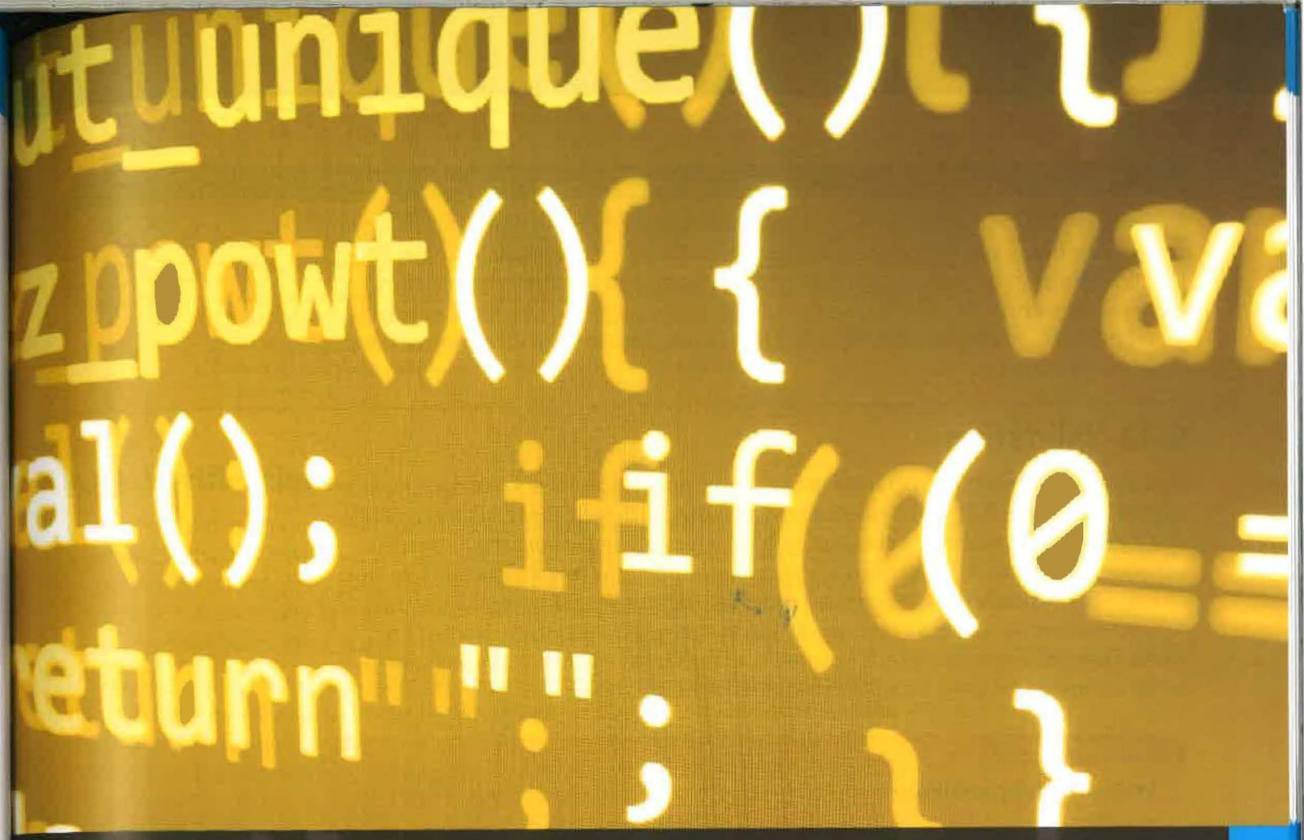
A user has been asked to investigate the numbers of students that are studying at schools in a geographical region. The user is required to input the name of the school and the number of its students for all 200 schools in the region. The system will output the name of the school with the highest number of students and the name of the school with the lowest number of students.

Task 7

A scientist is using a temperature sensor to record the exothermic reaction caused when two chemicals are mixed. Both chemicals start at the same temperature (which is recorded in the system). Once every minute, the sensor sends the temperature of the combined chemicals to the system. The system will continue to record the temperature values as long as the temperature remains above the initial start temperature. The expected output is the length of time in minutes the reaction takes to return to the initial value. It should also show the highest temperature reached and the time, in minutes from the start of the experiment, that this temperature was reached.

Summary

- Systems are made up of subsystems, which may in turn be made up of further subsystems.
- Top-down design is a method of simplifying the main system into its subsystems until the whole is sufficiently defined to allow it to be understood and programmed.
- Structure diagrams are a diagrammatical method of expressing a system as a series of subsystems
- When designing solutions for a given problem adopting an Input → Storage → Process → Output approach can help to design an effective solution.
- Standard methods of solution – flowcharts, pseudocode, top-down design, structure diagrams, simplification, efficiency analysis, validation and testing – are used when designing programmed systems.
- Effective solutions are ones that cannot be further simplified, are efficient, pass a testing regime and have effective validation of input.



Chapter 9:
Checking Inputs

Learning objectives

By the end of this chapter you will understand:

- the need for accuracy in inputs
- how to design validation routines using flowcharts or pseudocode
- the role and use of a range of validation techniques:
  - presence check
  - range check
  - length check
  - type check
  - format check
  - check digit
- how to program validation into your algorithms.

9.01 The Need for Accuracy

Organisations rely on the accuracy of their data when making decisions. Inaccurate data can compromise the validity of those decisions possibly with devastating results. Consider the situation of doctors receiving inaccurate medical data about patients or firefighters being given inaccurate data about wind speed and direction. The largest source of inaccuracies is during the data entry process. It is important that systems are designed to help increase the accuracy of data entry.

9.02 Validation

Validation is the process of programming a system to automatically check that data that falls outside a set of specified criteria is recognised as invalid. While validation cannot guarantee that data entered is accurate, it does ensure that it is reasonable. Systems should also filter out obvious mistakes. For example, if a system was recording the height of students, it would be reasonable to expect that they were all under 3 metres tall. Programming the system to reject data entries above 3 metres would help to remove obvious errors. However if a student's height was measured at 1.4 metres, but inaccurately entered as 1.04, the system would still accept the value because it meets the validation criteria.

SYLLABUS CHECK

Problem-solving and design: understand the need for validation checks to be made on input data.

TIP

Validation does not make data input accurate – this is a common misconception.

Table 9.01 shows different types of validation checks.

Table 9.01

Validation type	Description	Example
Presence check	Checks that required data has been input. The system will reject groups of data where required fields have been left blank. This is often used with data collection forms.	Online order where 'Email Address' must be provided.
Range check	Checks data falls within a reasonable range. Data outside the expected range is rejected. It is possible to have data where the range limit is only applicable to one extreme. For example, the volume of a vessel cannot be zero but may not have an upper limit. This is known as a 'limit check'.	Age must be between 0 and 130 inclusive. Day of the month must be between 1 and 31 inclusive. Percentage score in an exam must be between 0 and 100 inclusive.
Length check	Checks that data entered is of a reasonable length. Data items that have a length outside the expected values are rejected. Normally used with text-based inputs.	Surname must be between 1 and 25, inclusive, characters long. A password must have more than 6 characters.
Type check	Checks that a data item is of a particular data type. It will reject any input that is of a different type.	Stock items must be entered as an integer. Age will be numeric (e.g. it will not accept 'over 21').

Most important

Validation type	Description	Example
Format check	Checks that a data item matches a predetermined pattern and that the individual elements of the data item have particular values or data types.	Date of birth will be in the format dd/mm/yyyy. Mobile telephone number will be in the format NNNNN NNNNNN where N is a digit.
Check digit mod-11	Checks that a numerical data item has been entered accurately. Extra digit(s) are added to the number based on a calculation that can be repeated, enabling the number to be checked by repeating the calculation and comparing the calculated check digit with the value entered.	A barcode includes a check digit. ISBNs (book numbers) include a check digit.

9.03 Verification

Verification confirms the integrity of data as it is input into the system or when it is transferred between different parts of a system. Data integrity refers to the correctness of data during and after processing. Although the format of the data may be changed by processing, if data integrity has been maintained the data will remain a true and accurate representation of the original. Copying data should clearly not change the data values.

Most verification techniques are undertaken by the person inputting the data and involve the checking of the data input into the system against the original. One form of verification that you could program is 'double entry' verification. The data item is entered twice, often by different operators; the system compares the input values and identifies any differences.

Set up a new account Input password
Repeat the Input
Double entry can be done visually

9.04 Programming Validation into Your Systems

Running code without passing the correct values to the variables will cause a program to crash or provide unexpected results. Run any of your programs without inputting the required numeric values and you will receive an error message. This is illustrated in the following interactive session:

INTERACTIVE SESSION

```
>>> a = '1'
>>> b = 2
>>> c = a + b
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    c=a+b
TypeError: Can't convert 'int' object to str implicitly
>>>
```

This is important when getting user input of numbers in Python as the `input()` function always returns String data types. Python will not cast a String to a number without being explicitly asked to. Should this happen in a published program, the system would crash unexpectedly.

To avoid this type of error, validation should first try to cast the user input to a number with either `int()` or `float()`. If this process fails, the data entered should be rejected as a number of the correct sort was not entered.

school = input('Name of school: ')

Presence Check Validation

Code is required to check whether the user has input a value into a required field. If the data item is present, it will pass the presence check. This is no guarantee that the data item is in an acceptable format and additional validation may be required.

Figure 9.01 shows a flowchart for a simple presence check on user input in which a textual value is required.

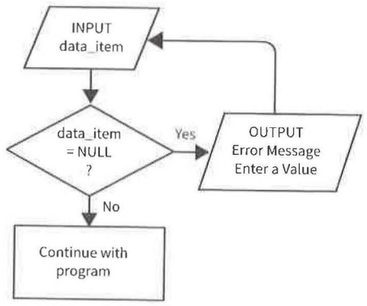


Figure 9.01 Presence check flowchart

In text-based programs, a WHILE loop can be used to check the presence of the input. If the data is missing, then the user can be prompted again to enter the required data.

```
data_item = input('Insert text value: ')

while data_item == '':
    print('Please input text value.')

# program continues
```

Easy to check

In GUI applications, the execution of code is triggered by an event such as a button press. The trigger cannot be controlled from within a WHILE loop in the same way. A different approach is therefore needed.

For comparison with the text-based code above, the following code provides a GUI implementation. A message is displayed in a label widget that previously held an empty String and so was not visible:

```
def button_click():

    if text_box.get() == '':
        my_label.config(text='Enter text here:')
    else:
        # program continues with for example:
        user_input = text_box.get()
```

Producing GUI based applications is outside the syllabus.

Note how this does not use a WHILE loop because the text box entry only has to be processed if the button is pressed. The tkinter `mainloop()` method, however, is running in its own loop that constantly checks for all events, including button presses.

Range Check Validation

Figure 9.02 shows a flowchart and pseudocode for a range check used to ensure the day of the month entered by a user is in the range 1 to 31. The code uses a WHILE loop that checks the data against the required criteria. If the data input is acceptable, the system will continue to run. If not, the system will output an error message to the user.

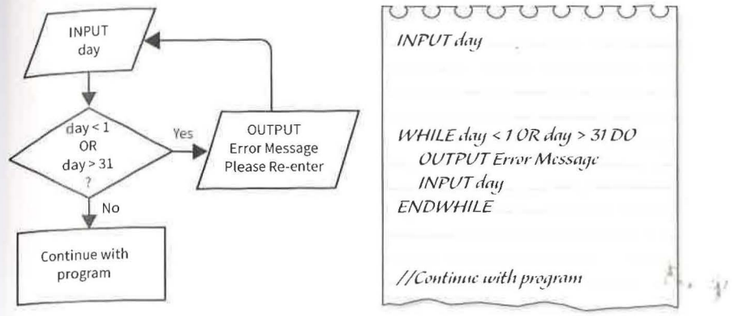


Figure 9.02 Flowchart and pseudocode for range check

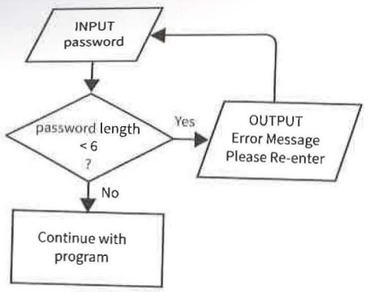
```
day = int(input('Enter the day of the month: '))

while day < 1 or day > 31:
    day = int(input('Enter a value between 1 and 31: '))

# Program continues
```

TIP

WHILE loops check criteria before running. The criteria are defined to identify inputs outside the required range. If an acceptable value is input, the loop never runs and the program continues. If an input is outside the expected range, the loop continues to iterate. This effectively halts the program until an acceptable value is input.

Length Check Validation

Figure 9.03 shows a flowchart and the pseudocode for a length check to ensure that a password consists of six or more characters. The code needs to calculate the length of the password. It then follows a similar process to a range check using a WHILE loop to check the input against the required criteria. If the data input is acceptable, the system will continue to run; if not, the system will output an error message to the user.




```
INPUT password

WHILE length of password < 6 DO
    OUTPUT Error Message
    INPUT password
ENDWHILE

//Continue with program
```

Figure 9.03 Flowchart and pseudocode for length check

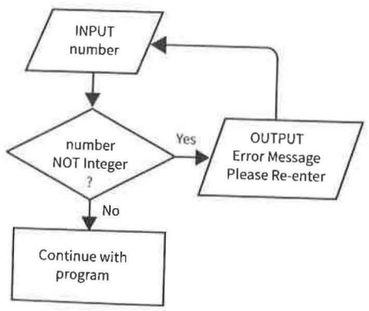
```
password = input('Enter password: ')

while len(password) < 6:
    password = input('Your password must have 6 or more characters: ')

# Program continues
```

TIP

Note the use of the `len()` function to return the number of characters in a String.

Type Check Validation

Figure 9.04 shows a flowchart and pseudocode for a type check. This ensures that a number is entered as an Integer. The code will need to identify the data type. It then uses a WHILE loop to check the input against the required criteria. If the data input is acceptable, the system will continue to run; if not, the system will output an error message to the user.




```
INPUT number

WHILE number IS NOT Integer DO
    OUTPUT Error Message
    INPUT number
ENDWHILE

//Continue with program
```

Figure 9.04 Flowchart and pseudocode for type check

While the pseudocode solution for type check validation is reasonably straightforward, the code approach is more complex. Python's `input()` function returns a String, so we would normally cast this to an Integer with the `int()` function. However, if the input is not an

Integer then the program will crash. What we want to do is evaluate the form of the input String. There is a function called `eval()` that we can use that does just this and then returns the appropriate type. Look at how this works in this interactive session:

INTERACTIVE SESSION

```
>>> number = input('Enter number: ')
Enter number: 2.1
>>> type(number)
<class 'str'>
>>> number = eval(number)
>>> type(number)
<class 'float'>
>>>
```

So one coded solution would be:

```
number = input('Please input a number: ')

while type(eval(number)) is not int:
    number = input('Not an integer. Please enter an integer: ')

# Program continues
```

A more advanced approach is to have a go at casting to an Integer and then use Python's built in `try...except` error handling code like this:

```
number = input('Please input a number: ')

while True:
    try:
        int(number)
        break
    except:
        number = input('Not an integer. Please enter an integer: ')

# Program continues
```

Notice how this solution puts the exception handling code in an infinite loop and then uses the `break` keyword to exit when the required conditions are met.

Format Check Validation

Figure 9.05 shows a flowchart and pseudocode for a format check to ensure that a date is in the format dd/mm/yyyy. The code will be required to check each element of the user input to ensure it matches a predetermined pattern. If the date matches the pattern, the system will continue to run; if not, the system will output an error message to the user.

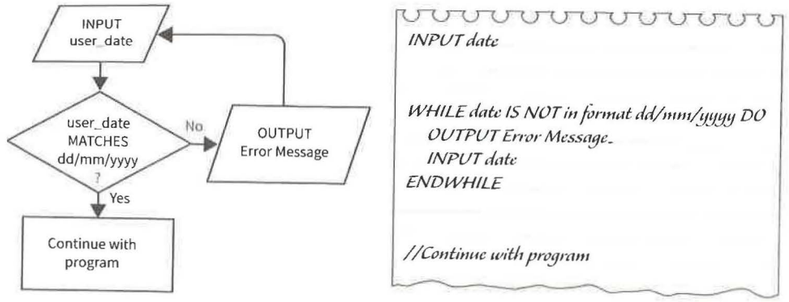


Figure 9.05 Flowchart and pseudocode for format check

As you might expect, Python has built-in libraries for validating a number of data types. As we are trying to ensure the integrity of this data, it is best, whenever possible, to use tried and tested methods. As this might well be a common requirement in a program, it might be best to implement the validation in a function and then call it. The code below imports the `datetime` methods from the `datetime` library and then implements the algorithm in the `validate_date()` function. A call for user input and the use of the function have been added to show how to use this function whenever a user is required to input a date.

```
from datetime import datetime

def validate_date(d):
    while True:
        try:
            return datetime.strptime(d, '%d/%m/%Y')
        except:
            d = input('Date must be in the format dd/mm/yyyy: ')

date = input('Please enter a date: ')
date = validate_date(date)

# Program continues
```

The `validate_date()` function tries to return a Python date data type that is now much safer and more flexible for the programmer to use than the initial String. If the `strptime()` function does not get passed a String in the pattern specified, an exception is thrown and the user is asked to try again. The `return` keyword is the equivalent of a `break` command but also returns a value. There are many patterns that can be asked for: `'%d/%m/%Y'` matches the requested format. More information can be found at: https://docs.python.org/3.5/library/datetime.html#strftime-strptime-behavior.

Check Digit Validation

Figure 9.06 shows a flowchart and pseudocode for an ISBN-13 check digit validation. In ISBN-13 validation, the 13th digit is removed as this is the check digit. All the other numbers are assigned a 1 or 3, alternating from 1. These numbers are used as multipliers for their corresponding digits. All the products are added together, the remainder after dividing this by 10 is found. The remainder is then subtracted from 10. This should match the check digit if the ISBN number is correct.

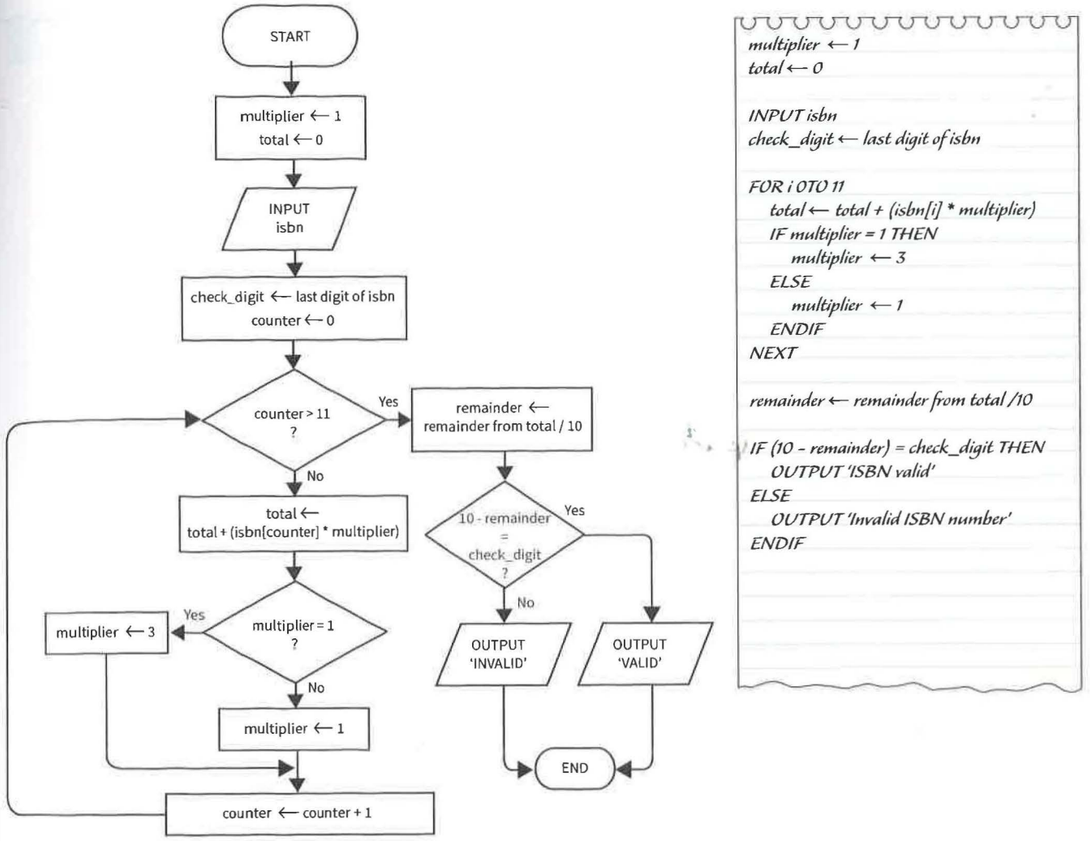


Figure 9.06 Flowchart and pseudocode for check digit validation of ISBN-13 numbers

The ISBN number, taken as input, is a pseudo-number. This is input as a string and each digit is accessed by the string index (starting from zero). This is discussed further in Chapter 11.

```
multiplier = 1
total = 0

# Input an ISBN-13 number as a string
isbn = input('Input an ISBN-13 number with no spaces: ')

# Obtain the 13th digit (all characters in string are numbered from zero)
check_digit = int(isbn[12])

# Iterate through the first 12 digits of the ISBN number
for i in range(12):
    total = total + (int(isbn[i]) * multiplier)
    if multiplier == 1:
        multiplier = 3
    else:
        multiplier = 1
```

```
remainder = total % 10

if (10 - remainder) == check_digit:
    print('ISBN valid')
else:
    print('Invalid ISBN number')
```

TASKS

Validation Tasks

Task 1

a Produce a flowchart and the pseudocode for a system that will only accept positive numbers.

b Code a text-based application for this algorithm.

Task 2

a Produce a flowchart and the pseudocode for a system that will only accept positive numbers less than 1000.

b Code a text-based application for this algorithm.

Task 3 (Optional)

Code a Python GUI application for a system that will only accept positive numbers less than 1000.

Producing GUI based applications is outside the syllabus.

Task 4

a Produce a flowchart and the pseudocode for a system that will check that users have input a password that is longer than six characters.

b Code a program for this algorithm.

Task 5

Program a function that will validate a user number and password from the user. All inputs must be validated against the following criteria:

- The user number must be a whole number in the range 1000 to 1500.
- The password must contain at least five characters.

The function will be passed the parameters of username and password and will return a Boolean value to indicate if the inputs match the validation criteria.

Task 6

Produce a flowchart and pseudocode for a system that makes use of the function in Task 5. If the validation is completed successfully the system will output 'Welcome'. If the validation check is failed twice the system will output 'Locked Out' and exit the program.

Hint: There are two ways of exiting a program in Python before reaching the end. First, you can wrap your main program in its own function such as `main()` and then call it at the end of your script – when you want to exit `main()` you use the keyword `break`. The second method is to import the `sys` module and then call `sys.exit()`.

Summary

- Accuracy of data entry is an important consideration in system design. Inaccurate data can lead to inaccurate outputs.
- Validation is a technique in which the system checks data input against a set of predetermined rules.
- Validation can identify obvious errors by detecting data that fails to meet the validation rules.
- Validation is able to ensure that data input is reasonable but cannot guarantee data accuracy.
- Six main forms of validation are used to check data as it is input:
  - presence checks ensure that data has been input
  - range checks ensure that data falls within a predetermined range of values
  - length checks ensure that data inputs contain a predetermined number of characters
  - type checks ensure that data input is of a certain data type
  - format checks ensure that data input meets a predetermined format, such as dd/mm/yyyy
  - check digits are calculated from numerical data such as a barcode and added to the end of the data.
- Verification checks the integrity of data when it is entered into the system. This is often completed by the individual inputting the data.
- Two common methods of verification are:
  - checking the input data against the original document or record
  - double entry in which the data is entered twice and the entries compared to identify differences.

Chapter 10: Testing

Learning objectives

By the end of this chapter you will understand:

- the importance of testing systems
- how to identify logical, syntax and runtime errors
- how to dry run algorithms using trace tables
- how to identify appropriate valid, invalid and boundary data when testing systems.

10.01 Why Test Systems?

In common with many products, it is important to make sure systems work as expected before they are released to the final user. The complexity and critical nature of the system will determine the extent of the testing to be completed. The computerised air traffic control system at an airport is more critical than a smartphone game and, therefore, will have undergone extensive testing – failure could be catastrophic.

There are several notable examples of disasters caused by poor testing. The destruction of the unmanned Ariane 5 space rocket due to the failure of untested code sequences is one of the most costly: The financial implications measured in billions of dollars. An article published in the *New York Times* magazine in December 1996 has more information about it and can be found at http://www.around.com/ariane.html.

10.02 When to Test

Testing can be broken into two distinct areas.

Alpha Testing

This is completed during the programming of a system to check that the individual code sequences work as expected before they are combined to make the complete system. Testing during the programming stage can also be completed to help trace the source of unexpected outcomes.

Beta Testing

Formal testing that takes place once the system has been completed to ensure that the whole system meets expectations.

10.03 Debugging

Debugging is the process of detecting faults that cause errors in a program. This can be achieved by observing error messages produced by the IDE or by investigating unexpected results. The types of error that can occur are divided into three groups.

Logical Errors

Logical errors are errors in the design of the program that allow it to run but produce unexpected results. They can result from the use of an incorrect formula or the incorrect use of control structures such as IF statements or loops. Examples include IF statements with incorrect conditions or loops that iterate the wrong number of times. Logical errors are also caused by implementing an incorrect sequence of statements, such as performing a calculation before assigning values to the variables.

Logical errors usually do not produce error messages. The problem is with the logic of the code not the execution of the code.

Syntax Errors

Syntax errors are errors in the use of the programming language such as incorrect punctuation or misspelt variables and control words. Examples include IF statements with missing colons or incorrect use of assignment. The IDE will usually generate error messages indicating the reason for the error.

Runtime Errors

Runtime errors are errors that are only identified during the execution of the program. They can result from mismatched data types, overflow or divide-by-zero operations.

Data type errors include:

- passing String data to an Integer variable, which will probably cause the system to crash.
- passing Real data to an Integer variable; the variable will round the input data to the nearest whole number – the system will execute the code but produce unexpected results.

Overflow errors occur when the data passed to a variable is too large to be held by the data type selected. In the theory element of the syllabus, you will have used this term to describe a situation where a nine-bit binary number is stored in an eight-bit byte. This can often result from calculations during the execution of a program. For example, in some programming languages the data type Short can be used to hold numbers between -32 767 and +32 767. If a variable of this data type was assigned the result of the square of any number greater than 182, it would produce an overflow error. As Python is a loosely typed language, it works behind the scenes to try and avoid many of these number type problems.

In mathematics, it is not possible to divide by zero because any number can be divided by zero an infinite number of times. If a program includes a division calculation that divides by a variable holding the value zero, the system will produce a divide-by-zero error.

INTERACTIVE SESSION

```
>>> 3/0
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    3/0
ZeroDivisionError: division by zero
>>>
```

SYLLABUS CHECK

Problem-solving and design: identify errors in given algorithms and suggest ways of removing these errors.

10.04 IDE Debugging Tools and Diagnostics

Many IDEs include sophisticated diagnostics designed to identify possible bugs and provide the user with supportive error messages. These tools are only able to identify errors in the code, not in the logic of the code, and as a result are unable to identify logical errors. IDLE and Wing IDE 101 provide debugging support for both syntax and runtime errors.

Syntax Diagnostics

Syntax errors can be spotted by noticing that the colour of your code in your IDE is not appropriate. For example, if you forget to close the quotation marks at the end of a String, the code will remain green in IDLE until the end of the line. IDLE has a helpful *Check Module*

command in the *Run* menu that will find the first instance of an error, highlight it in red and give you an indication of what the problem is, as shown in Figure 10.01.

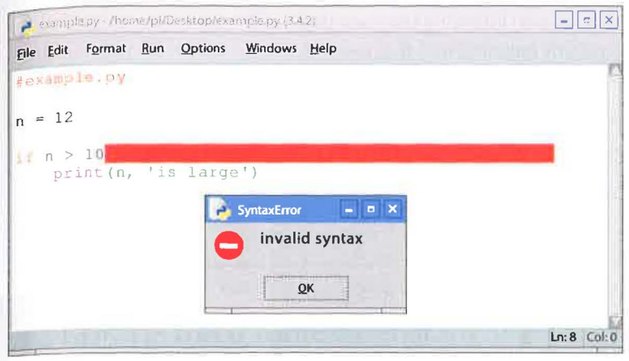


Figure 10.01 Error message in IDLE

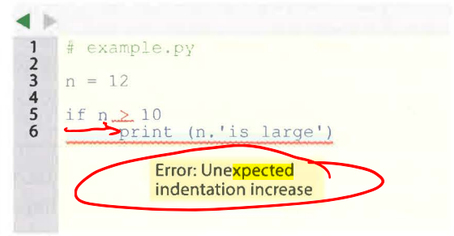


Figure 10.02 Error message detail in Wing IDE 101

In this instance, the colon is missing from the end of the line. In Wing IDE 101, this kind of error is shown underlined with a zig-zag line similar to the spell-checker feature in a word processor. Rolling the cursor over the error provides further information, as shown in Figure 10.02.

Runtime Diagnostics

Runtime errors are detected during execution of a program and specific error messages are provided. In Figure 10.03, the `input()` function returns a String variable, but the programmer has tried to use the input in a calculation. The IDE cannot detect that error – it would only occur at runtime (see Figure 10.03).

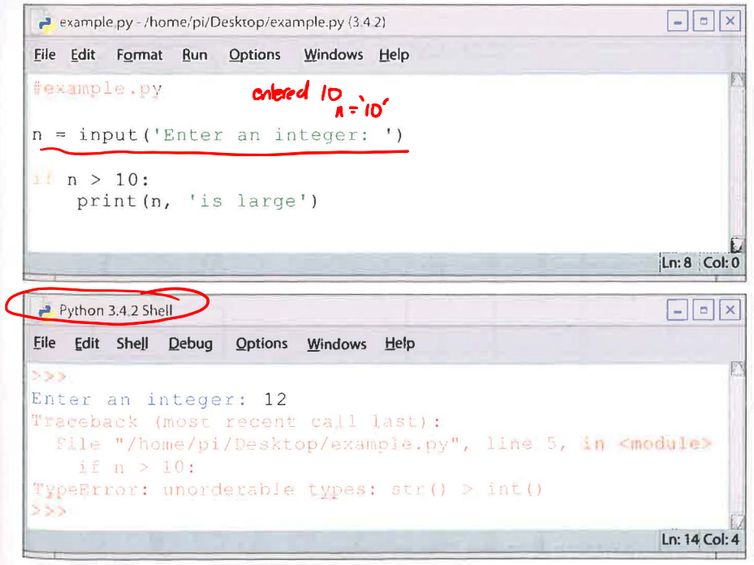


Figure 10.03 IDLE runtime error showing in Shell window

10.05 Identifying Logical Errors

While the IDE is able to support programmers with syntax and runtime errors, it cannot identify logical errors. The system will operate and process data by following the code that has been written – it is unable to determine if the code contains logical errors that result in unexpected outputs.

The process of identifying logical errors has to be part of the testing process. When unexpected outputs are recognised, it is likely that a logical error will be present in the code. The actual error will also have to be identified manually.

10.06 Dry Running

Dry running is the process of working through a section of code manually to locate logical or runtime errors. This type of testing often uses a trace table to record values within a system during its operation. The values traced could relate to the inputs, outputs or variables used in the process. It is usual to use a table with the variables listed as columns and their changing values recorded in rows.

SYLLABUS CHECK

Problem-solving and design: use trace tables to find the value of variables at each step in an algorithm.

Tracing Pseudocode

The following pseudocode algorithm is intended to calculate the quotient division (also known as Integer division) of x by y.

```
w ← 0
INPUT x
INPUT y

WHILE x > y DO
   x ← x - y
   w ← w + 1
ENDWHILE

OUTPUT w
```

DEMO TASK

Trace Table

Complete the trace table (Table 10.01) when the input values are x = 50 and y = 15.

Comments have been added to Table 10.01 to help explain the trace table. Comments are not normally required in a formal trace table.

Table 10.01

x	y	w	Output	Comments
		0		Initialisation value.
50	15	0		The new values are input.
35	15	1		x is reduced by 15, w is incremented by 1. ENDWHILE returns to the WHILE condition check. As x > y, the loop continues to run.
20	15	2		x is reduced by 15, w is incremented by 1. ENDWHILE returns to the WHILE condition check. As x > y, the loop continues to run.
5	15	3		x is reduced by 15, w is incremented by 1. ENDWHILE returns to the WHILE condition check. As x < y, the loop exits.
			3	The value in w is output.

EXTENSION TASK

Task 1 – Trace Table Extension

The pseudocode algorithm contains a logical error. Complete a trace table with the input values of x = 60 and y = 15 to identify the error.

Tracing a Flowchart

DEMO TASK

Flowchart Trace Table

Study the flowchart in Figure 10.04 and then complete a trace table (Table 10.02) where the input value is a = 2.

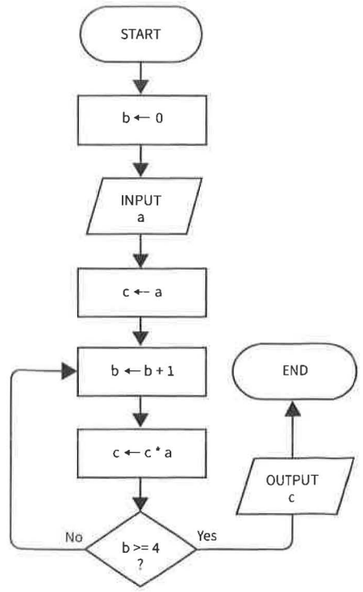


Table 10.02

a	b	c	Output	Comments
	0			Initialisation value
2	0	0		Input a
2	1	2		c ← a b ← b + 1 Will loop as b < 4
2	1	4		c ← c * a Will loop as b < 4
2	2	8		b ← b + 1 c ← c * a Will loop as b < 4
2	3	16		b ← b + 1 c ← c * a Will loop as b < 4
2	4	32		b ← b + 1 c ← c * a Loop exited as b = 4
2	4	32	32	Output value in c

Figure 10.04 Flowchart for a trace table

EXTENSION TASK

Task 2 – Discussion Question

a What is the aim of this flowchart?

b What kind of loop is being suggested here?

10.07 Breakpoints, Variable Tracing and Stepping Through Code

Although the IDE cannot identify logical errors, it does provide tools that assist the programmer in the manual process. IDLE and Wing IDE 101, in common with many IDEs, provide the programmer with the ability to execute the program one line at a time, displaying the values held in variables at each step. To allow the programmer to check particular segments of code, the system can be set to execute as normal until it meets a 'breakpoint'. These are created by the programmer, and will cause the system to run a line of code at a time.

In Wing IDE 101 this process is controlled by the buttons shown in Table 10.03.

Table 10.03

Button	Action
	Start or continue debugging until the next breakpoint is reached.
	Start debugging at the first line (or step into the current execution point).
	Execute the current line of code and then wait.
	Step out of the current function or method. (Useful if there is a long iteration present.)

The following algorithm has been designed to calculate the number of tins of paint required to cover a wall. The user inputs the length and height of the wall in metres and also the area that can be covered by one tin of paint. The algorithm does not produce the expected result.

```
length = int(input('Enter the length in metres: '))
height = int(input('Enter the height in metres: '))
coverage = int(input('How many square metres are covered by 1 tin? '))

area = length + height
tins = int(area / coverage)
```

The programmer decides to use the breakpoint diagnostic tool to help identify the error. The breakpoint is to be inserted after the input sequence as the programmer is happy that the correct inputs are being obtained. To test the system the programmer decides to use a length of 5 metres, a height of 2 metres and a coverage of 8 square metres per tin of paint; the expected area is 10 square metres.

To insert the breakpoint in Wing IDE 101 all that is required is to click with the cursor, next to the desired line number, in this case line 5. When the bug () is clicked, the code is executed in the normal fashion until the breakpoint is reached. The values of the various variables are viewable in the *Stack Data* panel. If a variable contains a value, hovering over the variable name will also show the value it currently contains.

To execute to line 5 and pause, the *step into current execution point* button () is clicked. This results in the screen shown in Figure 10.05.

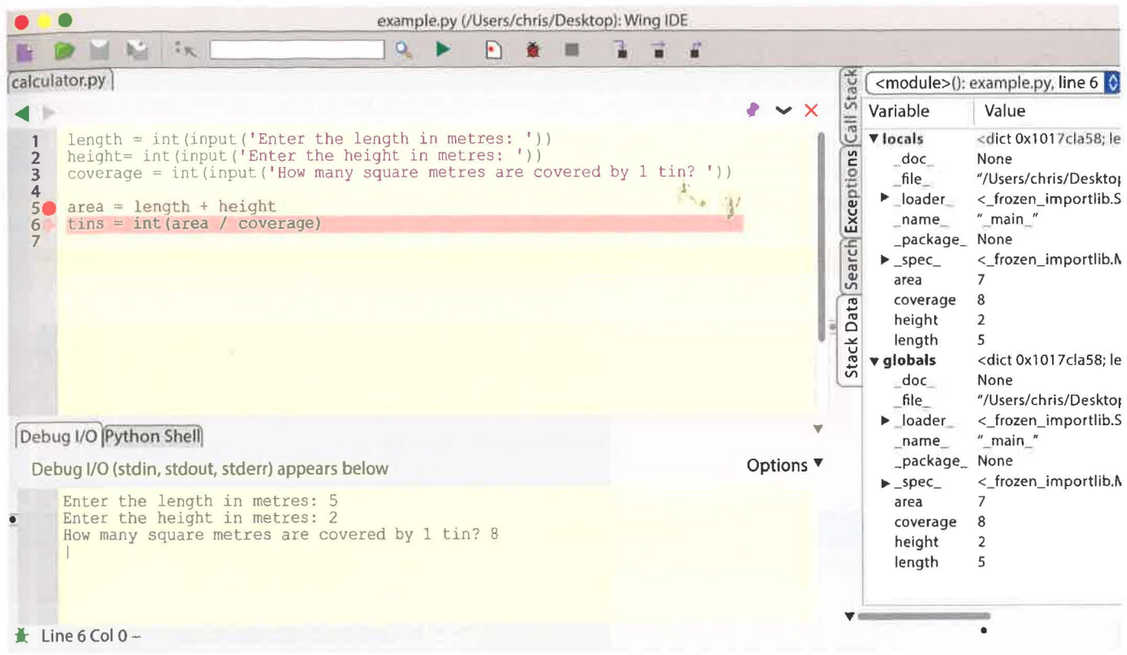


Figure 10.05 Wing IDE showing breakpoint variable values

The breakpoint is indicated by the red circle in line 5 and the next line to be executed is highlighted in pink. To delete a breakpoint click on it again.

In Figure 10.05, it can be seen that the area is 7, not the expected 10. This leads the programmer to identify the error in the calculation of area. They have incorrectly used addition (5 + 2 = 7), not the expected multiplication (5 * 2 = 10).

In IDLE, similar functionality can be achieved by following these steps:

- After writing your script and saving it somewhere, return to the Python Shell. The *Debug* menu is only available when the Shell window has focus (click on the Shell window to bring it to the front).
- From the *Debug* menu select *Debugger*.
- Return to the script file window and either *right-click* or *ctrl-click* on the line of code beginning with `area =` to create a breakpoint which will be shown by the line being highlighted.

- From the *Run* menu, choose *Run Module* or press *F5*, which will start executing the program with the *Debugger* on.
- Click the *Go* button in the *Debugger Control* window. The buttons on the *Debug Control* window function in a similar way to the buttons in Wing IDE 101. To keep track of the values in your variables, it is necessary to at least have the *Locals* and *Globals* checkboxes selected. See Figure 10.06 to see what the process looks like in IDLE running on a Raspberry Pi.

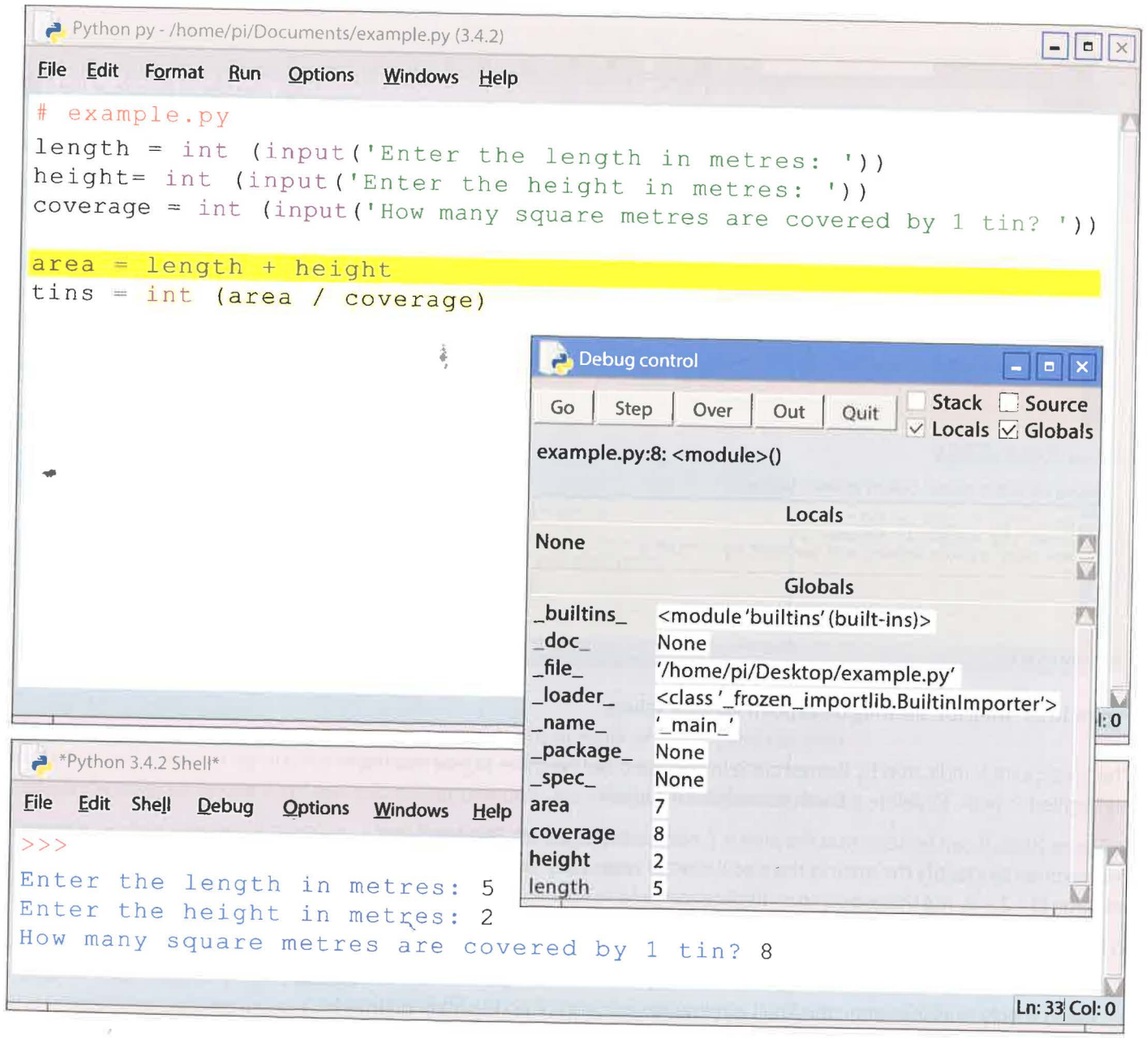


Figure 10.06 IDLE running with the Debugger on

TASK

Task 3 – Breakpoint

Once this error was fixed, the programmer continued to receive unexpected outputs for some test values.

a Copy the code into a script in your IDE and, using breakpoints and a range of data, identify the remaining error.

b Decide how you might correct this error.

10.08 Beta Testing

A formal test schedule is designed to test all possible events that a system could experience. It will test normal expected operation as well as extreme inputs or usage. The test schedule will identify the elements of the system to be tested and the data to be used in the tests. Each set of test data and the expected outcome is known as a 'test case'. The data used will fall into three categories, as described in Table 10.04. The example data in Table 10.04 is based on a system designed to determine the grade achieved by students in an examination, with inputs of student marks and the maximum possible mark.

SYLLABUS CHECK

Problem-solving and design: suggest and apply suitable test data.

Table 10.04

Type of test	Description	Example data
Valid data	Data that is expected to be met in the normal operation of the system. It meets the expected validation rules. The system should produce the expected outcome.	Integer values between zero and the maximum possible score.
Invalid data	Data that will not form part of the expected input range. The system should reject the data and output appropriate error messages.	Non-Integer values (it is not possible to get half a mark). Values less than zero or more than the maximum score. Textual inputs, such as 'TEN'.
Boundary data	Data that is at the boundary of the criteria that determine the path of execution of code.	Data that falls at grade boundaries. The grade boundary for an A is 80% and the maximum mark is 100. Data one mark below the grade boundary: 79. Data one mark above the grade boundary: 81.

TASK

Task 4 – Beta testing

A system holds the mobile phone number and age next birthday of current patients in a hospital. Mobile telephone numbers are entered as NNNNN-NNNNNN where N is a number. For each of the data items, decide on:

a the appropriate data type

b appropriate validation that could be applied

c invalid and, where appropriate, boundary data that could be used to test input validation.

Summary

- It is important to test systems to ensure they will perform as expected.
- Alpha testing is completed during the programming of a system.
- Beta testing is formal testing once the system has been completed.
- Logical errors are errors in the logic of the process performed by the code. The code will run but will produce unexpected results. These need to be debugged manually using, for example, trace tables.
- Syntax errors are errors in the syntax used within the code. It is likely that these will be identified by the IDE diagnostics.
- Runtime errors only become apparent during the execution of the code. Attempting to divide by zero is a common runtime error.
- IDEs identify syntax errors and runtime errors. They also provide useful tools to help debug logical errors by providing such facilities as the ability to step through code and add breakpoints.
- Trace tables provide a structure by which the value of variables, inputs and outputs can be traced at each step of an algorithm. They can be helpful in identifying logical errors.
- Valid data is met by the system in its normal operation.
- Invalid data is data that the system is not expecting. The system should identify and reject invalid data and provide appropriate error messages.
- Boundary data is data that falls at the boundaries of value changes. It is used to check the logic of the comparisons used to determine those value changes.

Chapter 11: Arrays

Learning objectives

By the end of this chapter you will understand how to:

- define an array using flowcharts and pseudocode
- declare and use an array
- read from and write values to an array
- use a number of arrays to organise data
- use Python's list data type as a substitute for arrays when implementing your algorithms.

11.01 What is an Array?

An **array** is a data structure that can hold a set of data items under a single identifier. Just as a variable holds data and is identified by a name, so an array also holds data of a specific type and is referred to by its name or label. While a variable can only hold one data value, an array can hold several values. For example, if you wished to store the surnames of 25 students you could either do this in 25 variables or a single array.

KEY TERMS

Array: A data type that can hold a set of data items of the same sort under a single identifier.

An array is an example of a container data type. Python does not support arrays out of the box. If arrays are required then the array built-in library must be imported first. This is because Python has a rich set of alternative container data types called tuples, lists and dictionaries. You will see later in this chapter that the Python version of the FOR loop is designed to work very closely with these container data types.

It is recommended that, when implementing your pseudocode and flowchart solutions in Python, you use lists to replicate arrays. A list can do everything an array can do and more. For example, lists are able to store a variety of data types at the same time and can be lengthened or shortened as required.

It is worth noting that, although lists are more flexible than arrays, arrays are more efficient for most simple processes.

11.02 Declaring an Array

Declaring an array is a similar process to declaring a variable. The difference is that you need to define the size of the array, determined by the number of data items that the array is required to hold. Each individual value held within an array is identified by an index number. Index numbers are sequential and, in Python, as with many other programming languages, the numbering starts from zero.

Table 11.01 is a diagrammatic representation of an array designed to hold the first five letters of the Greek alphabet.

Table 11.01

Index number	0	1	2	3	4
Data item	Alpha	Beta	Gamma	Delta	Epsilon

The pseudocode format for declaring an array capable of holding the five values is as follows. Note how the range of indices is 0 to 4:

```
greek_letter[0:4] OF String
```

Not all languages number array items from zero and you may see an example of arrays that start from 1. In this example the final index would then become 5:


```
greek_letter[1:5] OF String
```

When writing pseudocode, either method of numbering is acceptable; however, it is vital to remain consistent when using arrays in algorithms. If the array is declared as [0:4] then the following pseudocode must also follow that format with the first data item held in index 0 and the fifth data item in index 4.

SYLLABUS CHECK

Data structures: declare the size of one-dimensional arrays.

The syntax for declaring a list (the array substitute recommended for implementing your algorithms in Python) is:

```
my_list = [None]*4
```

Python's list container does not have to store a declared number of data items. This means that the syntax for creating an empty list of a given length does not look a lot like the pseudocode array declaration. The code above creates an array with four empty data spaces and is equivalent to

```
my_array[0,3]
```

in pseudocode.

If you are not sure how this works, start an interactive session and enter the line of code above. To see what has been created add the following line:

```
print (my_list)
```

FURTHER INFORMATION

Python programmers would normally just declare an empty list such as

```
my_list = []
```

to which data items can be added as required.

11.03 Initialising Arrays

At some point you will want to create an array that is ready filled with content. In pseudocode, it is achieved like this:

```
greek_letters ← ['Alpha', 'Beta', 'Gamma', 'Delta', 'Epsilon']
```

This is now a predefined array with five spaces that will only store Strings. In Python, the equivalent code would be:

```
greek_letters = ['Alpha', 'Beta', 'Gamma', 'Delta', 'Epsilon']
```

11.04 Using Arrays

Arrays offer programmers advantages over simple variables. As we have seen, they allow many data items to be stored under a single identifier. They give the programmer the ability to reference any individual data item by the appropriate **array index** and to use iteration to perform read, write or search operations by looping through the data items. This makes arrays particularly effective when working with data records.

KEY TERMS

Array index: A sequential number that references an item in an array.

SYLLABUS CHECK

Data structures: show understanding of the use of a variable as an index of an array.

Reading and Writing Data Items

To read a data item, you reference it by the array name and the index number. For example,

```
greek_letters[2]
```

holds the data item 'Gamma'. The same logic applies when writing values to an array. The following code would write the letter 'C' to the specified index position, replacing the original data item:

```
greek_letter[2] = 'C'
```

The syntax in Python for these two operations, using lists, is identical to the pseudocode.

DEMO TASK

Integer array

Declare an array named 'task' that is capable of holding four Integers. Write code to allow the user to input an Integer to selected array positions. Then add code to allow the user to output the value held in a selected array position.

Figure 11.01(a) shows the flowchart for the input process and Figure 11.01(b) shows the flowchart for the output process. Figure 11.02 shows the corresponding pseudocode for these processes.

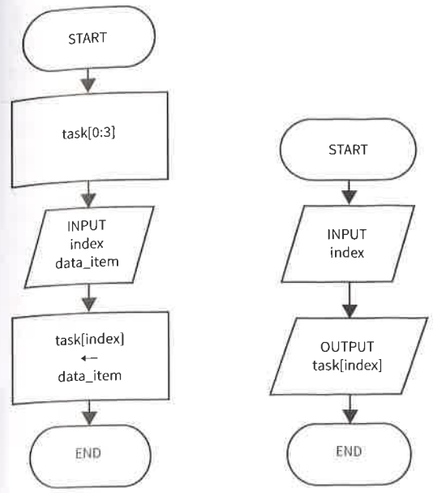


Figure 11.01 Array input (a) and output (b) flowchart

```
// Input Pseudocode
task[0:3]

INPUT index, data_item

task[index] ← data_item
```

(a)

```
// Output Pseudocode

INPUT index

OUTPUT task[index]
```

(b)

Figure 11.02 Array input and output pseudocode

A Python GUI application can be used to produce the system. The interface could be designed as shown in Figure 11.03. A button runs the subroutines for each of the input and output processes. `Entry` widgets accept the input and display the output. As the functions are so simple, the opportunity has been taken to demonstrate a little more tkinter GUI code.

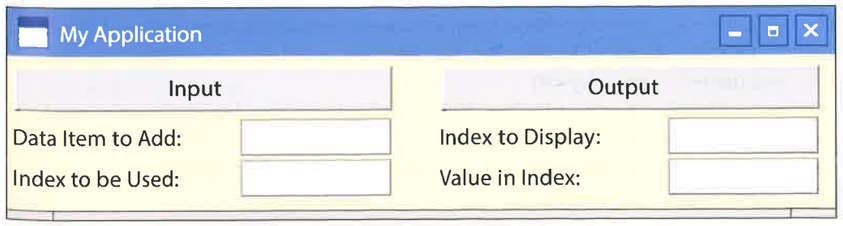


Figure 11.03 A GUI interface design

Producing GUI based applications is outside the syllabus.

```
from tkinter import *

# Declare list globally to allow both subroutines access to it
task = [None] * 4

# Functions
def input_data():
    # collect values from Entry boxes
    data_item = int(tbox1.get())
    index = int(tbox2.get())
    # insert new value into array
    task[index] = data_item

def output_data():
    # collect index required
    index = int(tbox3.get())
    # clear output text box and display value
    tbox4.delete(0, END)
    tbox4.insert(END, task[index])

#### Build the GUI
# padding is added to some widgets:
# ipadx adds internal padding to left and right
# padx adds external padding to left and right
window = Tk()
window.title('My Application')
bg_colour = 'linen'

# Create two frames
input_frame = Frame(window, bg=bg_colour)
input_frame.grid(row=0, column=0, ipadx=5, ipady=5)
output_frame = Frame(window, bg=bg_colour)
output_frame.grid(row=0, column=1, ipadx=5, ipady=5)

# Create the labels
input_label1 = Label(input_frame, text='Data Item to Add:', bg=bg_colour)
input_label1.grid(row=1, column=0, sticky=W)
input_label2 = Label(input_frame, text='Index to be Used:', bg=bg_colour)
input_label2.grid(row=2, column=0, sticky=W)
output_label1 = Label(output_frame, text='Index to Display:', bg=bg_colour)
output_label1.grid(row=1, column=0, sticky=W)
output_label2 = Label(output_frame, text='Value in Index:', bg=bg_colour)
output_label2.grid(row=2, column=0, sticky=W)

# Create the buttons
inputButton = Button(input_frame, text='Input', command=input_data, width=24)
inputButton.grid(row=0, column=0, columnspan=2, padx=5, pady=5)
outputButton = Button(output_frame, text='Output', command=output_data, width=24)
outputButton.grid(row=0, column=0, columnspan=2, padx=5, pady=5)
```



```
# Create the textboxes
tbox1 = Entry(input_frame, width=10)
tbox1.grid(row=1, column=1)
tbox2 = Entry(input_frame, width=10)
tbox2.grid(row=2, column=1)
tbox3 = Entry(output_frame, width=10)
tbox3.grid(row=1, column=1)
tbox4 = Entry(output_frame, width=10)
tbox4.grid(row=2, column=1)

# start tkinter loop
window.mainloop
```

EXTENSION TASKS

Reading from and Writing to an Array

Build the Integer array application and then try to write or read data with index 4. Try to input a textual value into the array. Both these actions will cause the system to crash and output an error message.

Task 1

What are the appropriate validation methods that could be used to prevent the user entering these types of invalid data?

Task 2

Draw a flowchart and create a pseudocode algorithm that includes these validation techniques.

Task 3

Test that your algorithm works by programming and running the code in Python and using suitable test data.

TIP

At some point, you may wish to know exactly what is in your array. In Python, all container data types can be printed to the console with a simple print statement. Sometimes it can be useful to insert such a print statement in your application's code when testing to print out the current state of your array. Look at this interactive session to see how it works:

```
>>> letters = ['a', 'b', 'c']
>>> print(letters)
['a', 'b', 'c']
>>>
```

Iteration in Arrays

The process of reading individual array positions can be extended by using a loop to read all the positions in an array. This allows iterative code to be used to check multiple data values.

Where the size of the array is known, a FOR loop can achieve the required iterative process. The counter variable in the FOR loop can be used to iterate through the index positions.

SYLLABUS CHECK

Data structures: read or write values in an array using a FOR . . . TO . . . NEXT loop.

The following pseudocode shows how iteration can be used to output all the data items in a ten-item array:

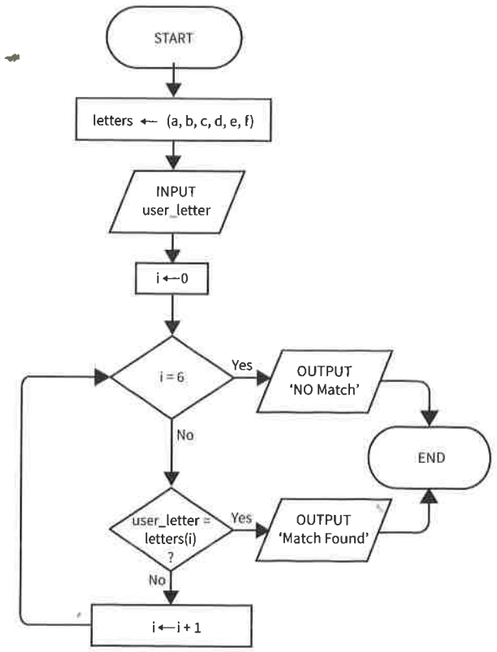
```
my_array[0:9]
FOR counter = 0 TO 9
    OUTPUT my_array[counter]
NEXT
```

DEMO TASK

Letters

Declare an array called 'letters' that is capable of holding six single characters. Initialise the array with letters a to f. Write code that allows the user to search the array to identify if any letter input by the user is in the array or not.

Figure 11.04 shows a flowchart and pseudocode for the algorithm.




```
FUNCTION search

    letters ← ('a', 'b', 'c', 'd', 'e', 'f')
    INPUT user_letter

    FOR i = 0 TO 5
        IF user_letter = letters(i) THEN
            OUTPUT 'Match Found'
            ENDFUNCTION
        END IF
    NEXT

    OUTPUT 'NO Match'

ENDFUNCTION
```

Figure 11.04 Flowchart and pseudocode for search algorithm

The FOR loop condition has been set to follow the declaration of the array, which has an index from 0 to 5. Once a match has been found the subroutine is ended. The 'No match' message is only shown if the loop has completed.

The following coded solution is a text-based implementation of the letters algorithm in Python. To illustrate how this works, a call to the function has been added after the function definition:

```
def search():
    # Initialise list to be searched
    letters = ['a','b','c','d','e','f']

    user_letter = input('Enter the character to search for: ')

    # Perform search of letters
    for i in range(0,6):
        if user_letter == letters[i]:
            print('Match found.')
            return

    print('No match.')

# Call function
search()
```

FURTHER INFORMATION (Optional)

The Python Way

There are many different programming languages. They all have their advantages and disadvantages. One of Python's strengths is the way it works with container variables. You may recall that the reason we use the `range()` function in the FOR loop is because Python FOR loops are of the iterative kind rather than the more traditional counter FOR loop. Python's FOR loop iterates through sequences or containers – `range()` creates a sequence of numbers. `letters` is already a container and so can be handled directly by Python's FOR loop. This kind of FOR loop is designed to loop through all items in the container irrespective of its length because lists, unlike arrays, can change length. The normal Python way of looping through letters would be like this:

```
# Perform search of letters
for letter in letters:
    if user_letter == letter:
        print('Match found.')
        return
```

Indeed Python is so adept at handling containers there is no need to use a FOR loop at all. Instead Python programmers would simply use the `in` keyword:

```
# Perform search of letters
if user_letter in letters:
    print('Match found.')
else:
    print('No match.')
```

Furthermore, as a String in Python is also a container data type, where all characters and spaces are indexed from zero, the following would also appear to function in the same way:

```
# Initialise string to be searched:
letters = 'abcdef'

search = input('Enter the character to search for: ')

# Perform search of letters
if search in letters:
    print('Match found.')
else:
    print('No match.')
```

Note that this is searching through a String data type and not an array. It is nice to know that all the methods you are learning about arrays are also useful in Python when manipulating String variables.

It would be a disservice to the reader, in a book on Python, not to point out its strengths. As a computer science student, however, you are required to iterate through arrays using a counter in a FOR loop. So when implementing your array algorithms, use the format in the solution shown above this further information box.

TASKS

Using iteration with an Array

The current system will stop once a match is found. This would be an ideal situation if the values are all unique, but this may not be the case.

For tasks 4 and 5, initialise the letters array with the following data items:

```
letters ← ('a', 'a', 'b', 'c', 'c', 'c')
```

Task 4

Draw a flowchart and create a pseudocode algorithm for a 'search' function that searches the entire array to check for multiple matches. The output should be the number of occasions a match was found.

Task 5

Test that your algorithm works by programming and running the code in Python.

11.05 Groups of Arrays

If you need to hold multiple data elements for each data record, it is possible to use arrays in groups. Provided that the same index number is used in each array for equivalent data items, multiple data items can be read. Consider a situation where a system holds the records shown in Table 11.02.

Table 11.02

Student ID	Surname	Computing grade
1001	Morgan	A
1002	Smith	C
1003	Jones	B

These data items could be held in three arrays as shown in Tables 11.03, 11.04 and 11.05 with the same index position in each array referring to the data regarding one record. As ID 1002 is held at index position 1 in the ID array, the remaining data for that record is held in index position 1 in the other two arrays.

Table 11.03 Student ID array

Index	0	1	2
Data item	1001	1002	1003

Table 11.04 Surname array

Index	0	1	2
Data item	Morgan	Smith	Jones

Table 11.05 Computing grade array

Index	0	1	2
Data item	A	C	B

The pseudocode to output the Surname and Grade for a given Student ID would be as follows:

```
INPUT search_ID
FOR i = 0 TO 2
    IF search_ID = ID[i] THEN
        OUTPUT surname[i]
        OUTPUT grade[i]
    ENDIF
NEXT
```

FURTHER INFORMATION (Optional)

Multidimensional Arrays and Other Containers

Programmers are not limited to one-dimensional arrays. It is possible to have arrays that can hold more than one set of data. It is worth pointing out that with Python's rich set of container data types, there are more elegant ways of solving the problem outlined at the beginning of this section. Indeed, the flexibility of Python's container data types and the libraries of methods that work with them is comprehensive. Lists, for example, can expand and contract; they can contain a mixture of data types including other containers. Dictionaries provide a kind of unordered list where the programmer chooses the key (instead of being limited to indexes of 0,1,2, etc.). These keys could be a String or number type variable and can contain a mixture of data types, including other containers. And of course, as discussed earlier, Strings can be treated as container data types in Python. By learning Python, you have a very powerful programming language at your disposal.

Both multidimensional arrays and Python's other containers are beyond the scope of the syllabus and this book. If you are interested in learning more about Python's container data types you might like to try one of the Coding Club level two books which explore Python's containers in a less formal way. (Find out more at http://codingclub.co.uk.)

11.06 Array Reference for Implementation in Python

Items can then be added using simple index calls:

Declaring an empty "array' of length 4:

Pseudocode:	Python code:

my_array[0:3] OF <data type>	`my_list = [None]*4`
	`print(my_list)`
	`output: [None, None, None, None]`

Items can be assigned using simple index calls:

Pseudocode:	Python code:
my_array[0] ← 2	`my_list[0] = 2`
my_array[1] ← 4	`my_list[1] = 4`
my_array[2] ← 6	`my_list[2] = 6`
my_array[3] ← 8	`my_list[3] = 8`

Items can be accessed using simple index calls:

Pseudocode:	Python code:

OUTPUT my_array[0]
Output: *2*
	`print(my_list[0])`
	Output: `2`
	`print(my_list)`
	Output: `[2, 4, 6, 8]`

Iterate through an array with a For Loop:

Pseudocode:	Python code:
FOR counter = 0 TO 3 THEN * OUTPUT my_array[counter]* *NEXT*	`for counter in range(0,4):` `  print(my_list[counter])`

Output:
2
4
6
8
	Output:
	`2`
	`4`
	`6`
	`7`

11.07 Array Tasks

TASKS

Task 6

a Draw a flowchart and create a pseudocode algorithm that iterates through an array of Integers and outputs the average. Declare and initialise the array with the following set of Integers: 12, 14, 10, 6, 7, 11 and 3.

b Test that your algorithm works by programming and running the code in Python.

Task 7

An algorithm will take an Integer value, *n*. It will call a subroutine to place into an array 12 incremental multiples of *n* (the first array index will hold $1 \times n$ and the last index position $12 \times n$). An additional subroutine will allow the user to output all the multiples in order.

a Draw a flowchart and create pseudocode for this algorithm.

b Test that your algorithm works by programming and running the code in Python.

Task 8

The data in Table 11.06 is to be organised in arrays so that the user can search via User ID and the system will display all the data related to that User ID.

Table 11.06

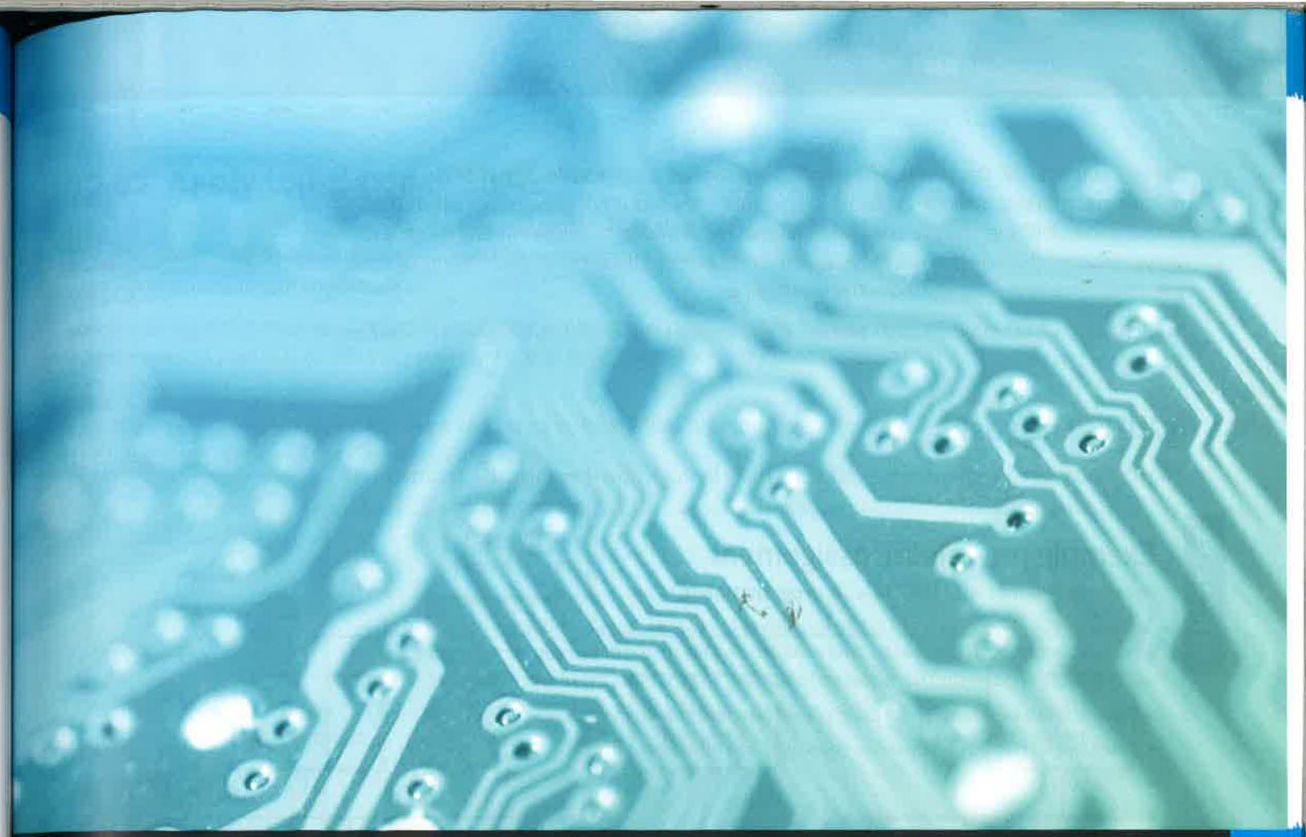
User ID	Age	Gender
112	45	Male
217	16	Female
126	27	Female

a Draw a flowchart and create a pseudocode algorithm that accepts a User ID and displays the related data.

b Test that your algorithm works by programming and running the code in Python.

Summary

- An array is a variable that can hold a set of data items, of the same data type, under a single identifier.
- When an array is declared, its size is defined. In Python indexes start from zero.
- Each element or data item in an array can be referenced by its index.
- The index can be used to read or write values in an array.
- A FOR loop can be used to iterate through the index locations in an array. The loop counter is used to identify successive index numbers.
- Holding records which consist of more than one data item can be achieved by the use of multiple arrays. Data for each record is held at the same index position in the different arrays.
- When using Python to implement algorithms involving arrays, a list is used as a substitute for an array.



Chapter 12:
Pre-release Task Preparation

Learning objectives

By the end of this chapter you will understand how to:

- work through your pre-release tasks for Paper 2
- combine top-down design and standard methods of solution to produce efficient algorithms
- combine all you have learnt about testing and validation to produce effective algorithms
- produce efficient and effective flowcharts, pseudocode and Python scripts