

# MY REVISION NOTES

## T-LEVELS

### DIGITAL PRODUCTION, DESIGN AND DEVELOPMENT

**T-LEVELS**  
THE NEXT LEVEL QUALIFICATION

# DIGITAL PRODUCTION, DESIGN AND DEVELOPMENT

- ✚ Plan and organise your revision
- ✚ Reinforce skills and understanding
- ✚ Practise exam-style questions

George Rouse



## 1 Problem solving

## 11 1.2 Algorithms

## 2 Introduction to programming

## 40 2.8 Testing

### 3 Emerging issues and impact of digital

## 51 3.2 Emerging trends and technologies

## 4 Legislation and regulatory requirements

## 64 4.2 Guidelines and codes of conduct

## 5 Business context

85 5.4 Risks in a business context

## 6 Data

## 99 6.4 Data management

## 7 Digital environments

128 7.5 Resilience of environment

## 8 Security

## 134 8.2 Threat mitigation

## 141 Glossary

143 Index

[illegible]

# 1 Problem solving

In order to solve a problem, it is necessary to understand it. Faced with a complex problem there is a range of techniques designed to change that problem into something that can be understood, formally represented and solved.

## 1.1 Computational thinking

Computer professionals use **computational thinking** techniques to solve real-world problems in order to create computer solutions. This does not require a computer but can be implemented on one. The same techniques can be applied to all manner of real-world problems, including those not involving computer programs.

Computational thinking is important because it makes it possible to understand and solve complex and **messy problems**.

**Computational thinking** A problem-solving technique developed by computer scientists that can be applied to real-world problems to make them understandable and solvable.

**Messy problems** Clusters of interrelated, interdependent complex problems, where it is difficult to provide a definitive statement of the problem. Examples might include pandemics, climate change, international crime activities and natural hazards.

### 1.1.1 Top-down, bottom-up and modularisation approaches to solving problems

REVISED

#### Top-down analysis

Top-down analysis attempts to break down a large problem into its component parts.

- ✚ The main problem is broken down into the main components.
- ✚ Each of the main components is analysed and broken down into a series of sub-problems.
- ✚ Each sub-problem is broken down further until an easily solvable solution is available.
- ✚ The top-down approach is mainly associated with structured programming languages such as C, Pascal and Python.

#### Key point

Python supports both the procedural, or structured, approach and the object-oriented approach to programming.

#### Bottom-up solution

This approach builds a solution by solving smaller problems and integrating them into a larger solution.

- ✚ The smaller parts of the problem should be easier to understand.
- ✚ By solving each of the individual elements it is possible to create a solution to the bigger problem.
- ✚ The bottom-up solution is mainly associated with object-oriented programming (OOP) languages such as C++, C# and Python.

## Modularisation

Top-down design and bottom-up solutions support the widespread use of modular programming (**modularisation**).

This has some advantages:

- ✚ The elements can be allocated to different programmers to speed up the process.
- ✚ Programmers can be allocated to solve those parts of the problem requiring specialist skills.
- ✚ **Debugging** the elements individually is more effective.

There are some drawbacks:

- ✚ This approach assumes that the whole solution is knowable in advance.
- ✚ With event-driven programming this neat top-down structure is not always appropriate.

However, the techniques can be applied to parts of the problem, and it remains a useful problem-solving tool.

**Modularisation** Separating the functional parts of a problem or program into independent modules that can be recombined to create a solution.

**Debug** To locate and fix errors in a program.

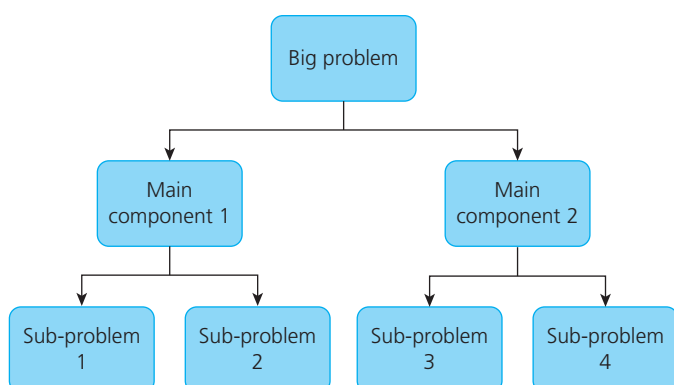
### 1.1.2 Solve problems using decomposition

REVISED

The top-down approach to problem solving uses **decomposition**. This is the process of breaking a problem down into smaller parts until each part can be easily solved.

In order to decompose a problem, the original problem needs to be understood and the main components identified. Each of these main components is then broken down into a series of sub-problems.

**Decomposition** Breaking down a complex problem into its component parts, which are easier to understand and solve.



**Figure 1.1** Decomposing a problem into its main components and sub-problems

Decomposition can be applied at various levels in computing scenarios, to the whole problem or to specific parts of the problem. We can break down a problem into different functional components that lead to:

- ✚ modules or program procedures
- ✚ processes, data stores and data flows.

The advantage of this approach is that:

- ✚ each sub-problem can be understood
- ✚ each sub-problem can be developed separately
- ✚ each sub-problem can be tested separately.

Combining these functional sub-problems should provide a solution to the original, more complex, problem.

There are some drawbacks:

- ✚ The original problem must be fully understood for this to produce a solution.
- ✚ The sub-problems may not provide a fully functional solution to the original problem.

### 1.1.3 Use pattern recognition

REVISED

Once a problem has been decomposed it is possible to identify patterns.

**Pattern recognition** is the process of looking for trends and similarities within and between problems or processes. It can be used to identify solutions that exist or can be modified, or to predict how a solution might be used effectively.

- + There may be existing solutions for some sub-problems that can be reused.
- + There may be existing solutions to similar problems that can be modified.
- + There may be a group of similar problems for which a common core solution can be created and modified.

The advantages of using pattern recognition include:

- + Existing code can be used; this means that a section of code works and is fully tested.
- + Existing code only needs to be modified to create a working solution, with only the new features needing to be tested.
- + The same basic code can be created for several elements and only needs to be tested once.
- + This newly created code, once tested, can be modified to create solutions for similar elements with only the modifications needing to be tested.

This saves time and improves reliability.

#### Pattern recognition

Looking for patterns or trends in order to identify potential solutions.

### 1.1.4 Use abstraction

REVISED

**Abstraction** is the process of taking a real-life problem and removing or hiding any unnecessary detail so that it can be analysed.

Abstraction is a representation of reality leaving just the key elements of the problem. This means the programmer can concentrate on the important aspects of the problem and not be distracted by unnecessary detail.

For example, a programmer may use a data structure without needing to know how it is implemented.

Large projects developed by teams often use elements developed by other programmers without needing to know how these work.

The advantages of this approach include:

- + The programmer can focus on the important aspects of the problem and ignore non-essential detail.
- + Teams of programmers can work on different aspects of the same problem.
- + Programmers can use pre-written or built-in functions without worrying about how they work.

A computer program is an abstraction of reality and when developing a program, we need to consider various things.

**Abstraction** The process of extracting the key features of a problem and ignoring the unnecessary detail.

#### Key features of the program

- + How will it be used?
- + Who will be using it?
  - + What is the skill set of the target user groups?
  - + What features are required by the target audience?

We need to think ahead and identify:

- + the 'output' from the system, including:
  - + on-screen information
  - + printed data
  - + stored data
  - + actions for the computer to complete

- + the input required to create the necessary output
- + the processes to turn the input into the required output.

When using abstraction, we create several layers, with each layer hiding the complexity of the previous one (**layering**). This means that only the detail required to solve one sub-problem is necessary for the programmer of that program element. Each layer should include the following.

- + What inputs are needed:
  - + including any validation and the data format.
- + What outputs are expected:
  - + including type, location (for example, on-screen, stored on disk, printed) and the format of that output.
- + What things will vary:
  - + **variables** (that is, the values that will change as a result of any input or process).
- + What things will remain constant:
  - + the **constant** values that must not be allowed to vary.
- + What key actions the program must perform:
  - + the processes to turn the input into the required output from the system.
- + What repeated processes the program will perform:
  - + the processes that are called more than once in the system.

**Layering** Organising a solution into segments that only interact with the previous and following segments.

**Variable** A value that will change as a result of another action.

**Constant** A value that does not change.

### Revision activity

Think about making a nesting box for birds.

- + Identify the elements/features required for a nesting box (top-down analysis).
- + Identify the raw materials required (decomposition).
- + Research plans for making the bird box and select appropriate elements to use or modify (pattern recognition).
- + Identify the unnecessary detail and discard it (abstraction).
- + Write a brief set of details for how to make the bird box.

### Now test yourself

TESTED ☐

- 1 What is computational thinking?
- 2 What are the advantages of the top-down approach?
- 3 What is meant by pattern recognition?
- 4 What are the advantages of using a modular approach (modularisation)?
- 5 Why is layering an advantage for the programmer?
- 6 Why might decomposition not provide a solution to a problem?

## 1.2 Algorithms

### 1.2.1 Algorithms, what they are and how they are expressed

REVISED ☐

Computers are only able to solve problems if they are given the right set of instructions to follow. An **algorithm** is a set of step-by-step instructions that describe solutions to problems.

Algorithms can be expressed in terms of flowcharts, written descriptions, pseudocode and program code.

**Algorithm** A set of instructions describing a solution to a problem.

## 1.2.2 Expressing an algorithm using flowcharts and pseudocode, and the use of these when planning a digital solution

Writing algorithms that describe a solution to a problem can be a complex process, which is why we often use computational thinking to derive them.

An algorithm must:

- + be precise enough to define a problem accurately
- + contain all the necessary steps in the process
- + identify clearly the instructions that need to be followed and in what order
- + identify what decisions need to be made, when they need to be made, and what to do based on the result of those decisions.

The advantages of using algorithms include:

- + They don't need specialist knowledge to understand the processes.
- + They provide a detailed breakdown of the solution.
- + They provide an outline for a coded solution.
- + They are language independent.

The disadvantages include:

- + Some constructs can be quite difficult to represent.
- + They can be quite time-consuming to create.
- + An algorithm can be expressed in several ways.

There are several ways to represent algorithms. Written descriptions are often the starting point when creating an algorithm.

Algorithms are written as a set of descriptors or instructions in standard English, requiring no formal **syntax** or structure. This means anyone can create an algorithm using a set of brief descriptors.

A written description of an algorithm:

- + should use short sentences or statements
- + does not require specific details of how the solution will be coded
- + must include all the parts of the solution to the problem.


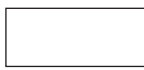
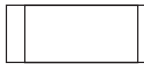
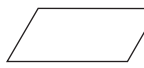
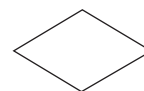

It can, however, be quite difficult to express some technical structures in detail using written descriptions, particularly decisions and the follow-up actions and **loops**, so graphical representations such as flowcharts may be used instead.

**Flowcharts** use standard shapes to represent the various features of an algorithm.

**Syntax** The formal structure for a statement.

**Loop** A sequence of commands that is repeatedly executed until a condition is reached.

**Flowcharts** Diagrammatic representations of algorithms.

	Line	An arrow represents control passing between the connected shapes.
	Process	This shape represents something being performed or done.
	Subroutine	This shape represents a subroutine call that will relate to a separate, non-linked flowchart.
	Input/output	This shape represents the input or output of something into or out of the flowchart.
	Decision	This shape represents a decision (Yes/No or True/False) that results in two lines representing the different possible outcomes.
	Terminal	This shape represents the 'Start' and 'End' of the process.

**Figure 1.2** Standard flowchart shapes

### Revision activity

Write a description for a program that checks a password for length, for the inclusion of numbers, and for upper- and lower-case letters.

- + All flowcharts start and end with the terminal shape.
- + Inputs and outputs are represented by a parallelogram.
- + Decisions are represented by a diamond shape with two outgoing paths: Yes/No or True/False.
- + Processes are represented by rectangles.
- + The symbol to call a **subroutine** is a rectangle with two vertical lines.

Flowcharts have some advantages:

- + Using a standard set of symbols means they can be easily followed and understood by many people.
- + The clear structure is able to show the flow through a program.

They also have some disadvantages:

- + Being graphical they can become large very quickly, making complex problems difficult to represent and follow.
- + Any modifications require a significant amount of work to amend or even redraw the diagram.

**Pseudocode** is a technical representation of an algorithm. The choice of flowchart or pseudocode depends upon the audience requirements. Flowcharts are useful for communicating between technical and non-technical people, pseudocode between technical people and programmers. It is often the next stage in the process, turning the flowchart into a coded solution. While it does not use any programming language syntax, it is written in a format similar to a high-level language.

The advantages of pseudocode include:

- + The structure makes it relatively easy to convert pseudocode to program code.
- + It can be modified quite easily (unlike a flowchart).
- + It is reasonably easy to follow.

The disadvantages include:

- + It can be a time-consuming process.
- + In some cases, it can be hard to follow data flows.

Program code is the code that can be executed on a computer, but if the person writing the algorithm has the necessary coding skills, they might choose to write the pseudocode in a draft version using their chosen programming language.

This has some advantages:

- + It shortens the step from pseudocode to code.
- + Syntax errors are not important at this stage and can be corrected at the coding stage.
- + All the program constructs and built-in functions can be used.

There are some disadvantages:

- + The process may end up trying to write the program code without fully analysing the problem, meaning it does not lead to an effective solution to the problem.
- + The algorithm will not be language independent.
- + Some knowledge of the language will be required to understand the algorithm.

### 1.2.3 Algorithms for the key programming constructs

There are three standard constructs used in computer programs:

- 1 Sequence
- 2 Selection
- 3 Iteration.

**Subroutine** Code that can be called from within the program. After the subroutine completes, control is returned to the main program.

#### Revision activity

Use your written description of the password program to create a flowchart to describe it.

**Pseudocode** A more structured form of English used to describe algorithms.

#### Revision activity

Turn the flowchart you created into pseudocode.

#### Typical mistake

In this case program code does not refer to code that can be executed but merely a more precise form of pseudocode based on a specific language.

## Sequence

**Sequence** executes the instructions one after the other.

Note how the order in the two programs below is important. The programs produce different outputs.

### Worked example

#### Program 1

```
x = 9
x = x + 6
x = x / 3
print(x)
```

#### Program 2

```
x = 9
x = x / 3
x = x + 6
print(x)
```

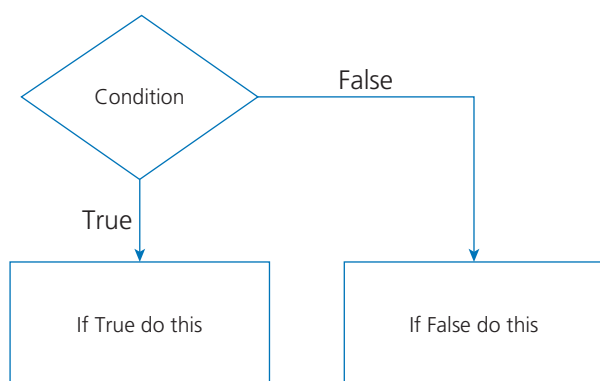
**Sequence** Code that executes one instruction after another.

**Selection** The code to execute is determined by the outcome of a condition.

**Iteration** (often called a loop) The code executed depends on a condition being met or executed a set number of times.

## Selection

**Selection** is the construct that decides how to proceed based on the result of a decision (Figure 1.3).



**Figure 1.3** Selection based on the result of a decision

### Worked example

A simple algorithm might be:

```
Set a value for num1
Set a value for num2
If num1 is bigger than num2
Print num1 is bigger
```

In the specification pseudocode:

```
RECEIVE num1 FROM (INTEGER) KEYBOARD
RECEIVE num2 FROM (INTEGER) KEYBOARD
IF num1 > num2 THEN
    SEND 'num1 is bigger' TO DISPLAY
END IF
```

In Python, the code for this is:

```
num1 = int(input())
num2 = int(input())
if num1 > num2:
    print('num1 is bigger')
```

In Python you must indent code within a loop or a conditional statement. It is also good practice to do this when writing algorithms, especially in pseudocode. By indenting the code within these structures, it makes it much easier to see what code is being repeated or relates to the conditional statement.

### Exam tip

In the examination you should expect to see questions using pseudocode to be written using the specification style. Unless specifically told to use the specification pseudocode style, any responses in a more general pseudocode should be acceptable, providing they clearly describe the solution.

## Iteration

**Iteration** is a construct that repeats a process either until a condition is true, while a condition is true, or a set number of times.

**Repeat until** executes a code segment until the value of a condition is true (Figure 1.4).

The structure means that a repeat until condition must execute at least once since the condition is not checked until one iteration has been completed.

### Worked example

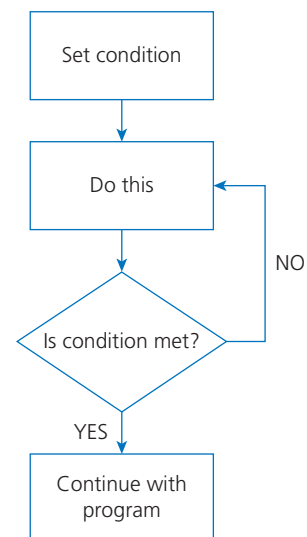
In pseudocode we can create an algorithm to print the values 1 to 5:

```
x = 1
repeat
  print(x)
  x = x + 1
until x = 6
```

In the specification pseudocode:

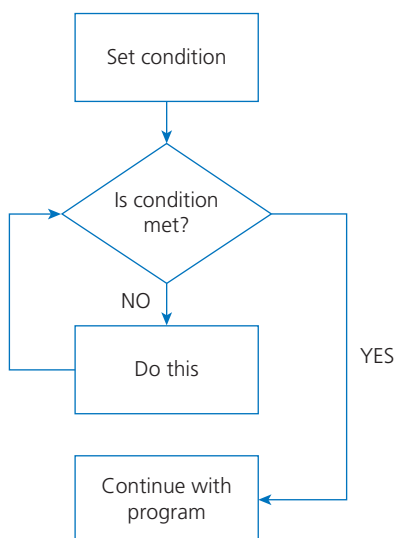
```
SET x TO 1
REPEAT
  SEND x TO DISPLAY
  SET x TO x + 1
UNTIL x = 6
```

Note that Python does not support the repeat until structure.



**Figure 1.4** A repeat until loop – note that the loop executes until a condition is met

**While** is a construct that executes a code segment while a condition is true.



**Figure 1.5** A while loop – note that the condition is checked before any code can execute

In this structure the condition is checked before any code in the loop is executed and might not execute at all if the condition is met initially.

**Worked example**

In pseudocode the algorithm becomes:

```
x = 1
while x < 6
    print(x)
    x = x + 1
endwhile
```

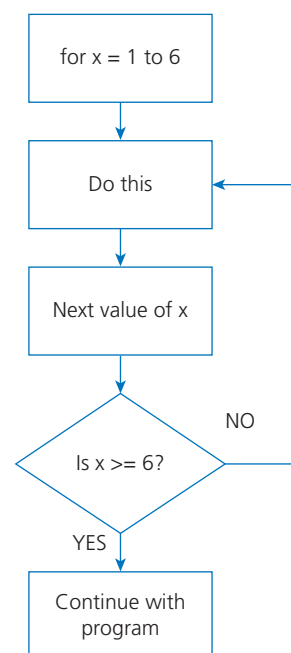
Python does support the while construct:

```
x = 1
while x < 6:
    print(x)
    x = x + 1
```

In the specification pseudocode:

```
SET x TO 1
WHILE x < 6 DO
    SEND x TO DISPLAY
    SET x TO x + 1
END WHILE
```

**Increment** The process of increasing a numeric value by another value, often 1. For example, the sequence 1,2,3 ...



**Figure 1.6** A for next loop will execute a fixed number of times, exiting the loop once the index value reaches the end of the specified range

**Worked example**

In pseudocode we can create an algorithm to print the values 1 to 5.

```
for x = 1 to 5
    print(x)
next x
```

In Python:

```
for x in range(1,6):
    print(x)
```

In the specification pseudocode:

```
FOR x FROM 1 TO 5 DO
    SEND x TO DISPLAY
END FOR
```

**Revision activity**

Turn your pseudocode algorithm into a coded solution.

Note how the range is 1 to 6, not 1 to 5. If you look at the flowchart you can see that once the value reaches 6 it does not execute the loop again. In the Python range it executes the loop with values up to one less than the top of the range.

## 1.2.4 Understand the purpose for a given algorithm and how the algorithm works

REVISED

There may be several different ways to approach the same problem, generating a number of different algorithms. The best way to determine what the algorithm does, the output it produces and if there are any errors in it is to dry run it.

## 1.2.5 Determine the correct output of an algorithm

REVISED

Dry running an algorithm can be as straightforward as simply following it through line by line to determine the purpose. For a written description, turning it into pseudocode or a flowchart might provide more insight.

A more rigorous approach would be to use a **trace table** to follow through what happens to each variable as the program executes.

A trace table consists of a table with a row for each line of code and a column for each variable. By following the algorithm through line by line, the values of each variable are recorded at each stage.

**Trace table** A table used to track the state of variables in an algorithm or program, to check how it works and that it works as expected.

### Worked example

For the pseudocode:

```
RECEIVE num1 FROM (INTEGER) KEYBOARD
RECEIVE num2 FROM (INTEGER) KEYBOARD
IF num1 > num2 THEN
    SEND 'num1 is bigger' TO DISPLAY
END IF
IF num2 > num1 THEN
    SEND 'num2 is bigger' TO DISPLAY
ELSE
    SEND 'both numbers the same' TO DISPLAY
END IF
```

The trace table needs eight rows for the eight lines of instructions, and columns for line number, num1, num2, results of condition, output and a comment.

The trace table would be used to check a range of possible inputs to see if it works as required:

- + one set with num1 bigger than num2
- + one with num2 bigger than num1
- + one with num1 and num2 the same value.

The result when num1 is 6 and num2 is 7 is:

Line number	num1	num2	Condition	Output	Comment
1	6				
2	6	7			
3	6	7	False		
6	6	7	True		Lines 4 and 5 skipped
7	6	7	True	num2 is bigger	Program ends

### Revision activity

Complete the trace table in the Worked example with values for num1 bigger than num2 and num1 the same value as num2.

## 1.2.6 Identify and correct errors in an algorithm

REVISED

You can check an algorithm by using a trace table or by following a flowchart. Using a range of values for the variables, or otherwise, you can identify how the algorithm works - more importantly, if it does not do what it is supposed to.

If an error is identified, the algorithm should be modified and checked again.

Remember to include comments when writing algorithms, to explain each section of code, so that when it is checked for errors or needs to be modified, there is no confusion about what the algorithm is meant to do.

### Now test yourself

TESTED

- 1 What is an algorithm?
- 2 What is meant by the term 'syntax'?
- 3 State two disadvantages of representing algorithms using flowcharts.
- 4 Describe the differences between a repeat loop and a while loop.
- 5 Why would you dry run an algorithm?
- 6 Why do we use comments in an algorithm?

### Summary

In this chapter you learned about:

- + Computational thinking
- + Top-down, bottom-up and modularisation approaches to solving problems
- + Solving problems using decomposition
- + Using pattern recognition
- + Using abstraction
- + Algorithms
- + What algorithms are and how they are expressed
- + Expressing an algorithm using flowcharts and pseudocode, and the use of these when planning a digital solution
- + Writing algorithms for the key programming constructs
- + The purpose of a given algorithm and how the algorithm works
- + Determining the correct output of an algorithm
- + Identifying and correcting errors in an algorithm

### Exam practice

- 1 Explain why computational thinking is used to design solutions to complex problems. [6]
- 2 State and describe two computational thinking principles and how they are used when developing a computing solution to a problem. [6]
- 3 A local transport organisation is designing a map for the local bus routes and interchanges.
  - a Explain what is meant by abstraction. [2]
  - b Identify two things from the real world that will remain in the map. [2]
  - c Identify two things from the real world that will be missing from the map. [2]
- 4 Three main ways of writing an algorithm are: written in plain English, flowcharts and pseudocode. Discuss the relative merits of these three approaches. [8]
- 5 An employer wants to know how its staff travel to work in the mornings. The employer needs a program to ask each member of staff if they travel by bus, train, bicycle, car or walk. The program should then output the totals for each mode of transport. Write an algorithm in any form of pseudocode to represent this program. [6]

- 6 There are three major ways to create an iterative loop in a program.
- a Describe how a repeat loop works in a program. [3]
  - b Describe how a while loop works in a program. [3]
  - c The repeat loop described by the pseudocode below adds together three numbers input by the user and prints the result. Modify this to use a while loop. [4]

```

SET count TO 1
SET total TO 0
REPEAT
    RECEIVE num FROM (INTEGER) KEYBOARD
    SET total TO total + num
    SET count TO count +1
UNTIL count > 3
SEND total TO DISPLAY

```

- 7 A programmer is writing a program to calculate the average of a set of values input by the user. The user should be asked to input a value, then asked if they wish to add another one. If they answer Yes, then the program asks for another value, adds it to the total and adds one to the count of values added. This repeats until the user enters No. At this point the program divides the total by the count and prints the average value. Using a flowchart, write an algorithm to describe this process. [6]
- 8 The algorithm below is part of a hangman game. It is supposed to ask for an input from the user to guess a letter that might be in the word. It then checks each letter in the word and prints found at position x or Not found. Using a trace table, dry run this algorithm using the word car and the input a. [8]

```

RECEIVE word FROM (STRING) KEYBOARD
RECEIVE guess FROM (STRING) KEYBOARD
SET flag TO 'Not found'
FOR index FROM 1 TO length of word DO
    IF guess = letter in word at position index THEN
        SEND guess, 'found at position', index TO DISPLAY
        SET flag TO 'Found'
    END IF
END FOR
IF flag = 'Not found' THEN
    SEND 'Not found' TO DISPLAY
END IF

```

# 2 Introduction to programming

Programming is about turning algorithms into coded solutions that will run on a computer to solve a problem. The computer will only do as it is instructed, so program code must be accurate. Programs are written using a suitable programming language. High-level languages are meant to be understandable to humans and must be translated into low-level machine code for the computer to be able to execute them. There are many programming languages, the best choice being a language suited to the task. In this book we will refer to the language Python, which is an easy-to-use general-purpose language.

## 2.1 Program data

### 2.1.1 The use of, and need for, data types

REVISED 

Computer programs process data. The type of data depends on what processing is required to turn inputs into useful outputs, and any storage requirements. There are five main types of data:

Data type	Description	Example
Integer	Whole numbers with no fractional or decimal parts	99, -55, 3
Real (floating point)	Numbers that can have a decimal or fractional part	3.72, -1.87, 2.5
Character	A single alphanumeric character, including symbols	S, f, @
String	A collection of alphanumeric characters	C&g, 5tH, 'Hi there'
Boolean	Can only take one of two values: True or False	True, False

### 2.1.2 Declare and use constants and variables that use appropriate data types

REVISED 

In some programming languages it is necessary to declare the type of data being allocated to a variable. In Python this is not necessary, and any variable allocated a value will automatically be set to the appropriate type.

Similarly, some programming languages require constants to be declared, but this is not supported in Python.

#### Exam tip

To gain the most marks, remember that variables are values that can vary during the execution of a program, and constants are values that do not change during the execution of a program.

The concept of a constant is quite useful for values that will not change, such as VAT rates or discount rates.

#### Typical mistake

It is a mistake to describe a constant as a variable that does not change. There are similarities, but they are different.

**Cast** To allocate a data type to a variable.

It is possible in Python to **cast** a variable to a specific type; this is important when getting user input.

Python uses:

- + `int()` to set a variable to integer or return an integer value from a variable or value
- + `float()` to set a variable to real or return a real value from a variable or value
- + `str()` to set a variable to string or return a string value from a variable or value
- + `bool()` to set a variable to Boolean or return a Boolean value from a variable or value.

To force an input value to be float, for example, the instruction would be:

```
value = float(input('Enter a number '))
```

Casting variables to another type:

Expression	Value
<code>x = str(3.25)</code>	'3.25'
<code>x = int(3.25)</code>	3
<code>x = bool(3.25)</code>	True
<code>x = int('7')</code>	7

Of course, not all data can be cast to another type. For example, `int('Three')` is meaningless and will generate an error.

## 2.1.3 The use of, and need for, data structures

REVISED

A variable can be used to store a single item of data within a program, but large quantities of data can be managed by using a data structure to store multiple items using a single **identifier**.

- + This enables the programmer to manage data in a more efficient way.
- + It makes following and understanding the program significantly easier than it would be with multiple identifiers.
- + The data structures can be reused with different data requiring the same structure.
- + Searching for or storing data in a suitable data structure is much simpler than working with multiple identifiers.

There are three common data structures that are fundamental features in many programming languages:

- 1 List
- 2 Array
- 3 Dictionary.

### List

**Lists** use a single identifier and an **index** to create a data structure. A list:

- + is an ordered structure
- + is accessed through an index that indicates the position of an item in the list
- + has no predefined **scope**
- + does not require defined attributes, making it easier to initialise
- + can be accessed in index order using a very straightforward program
- + is not limited to a single data type.

#### Worked example

For this list of names:

Index	0	1	2	3	4	5	6
Data	'Jones'	'Ahmed'	'Lee'	'Fletcher'	'Khan'	'Brown'	'Chen'

```
names[2] = 'Lee'
```

**Identifier** The name given to the data structure; together with an index value it can identify a single item in the list.

**List** An ordered set of data using a single identifier and an index to identify a data item.

**Index** The position of a value in a list; note that in Python we start counting at 0.

**Scope** The extent to which a variable can be seen by the different parts of the program.

**Array** A data structure using a single identifier and multiple indices to identify a data item.

We could access this by using a simple loop, a for loop using the length of the list as an end point for the range.

## Array

An **array** is a data structure that allows the programmer to use a single identifier and multiple indices to store data. An array:

- + can be one-dimensional using a single index, or multi-dimensional using multiple indices
- + has a predefined scope
- + can only hold data of the same type
- + uses indices so it is easy to search for data or sort the data.

### Worked example

For this list of stock in a shop (stock):

Index	0	1	2	3
0	'Orange'	'Apple'	'Banana'	'Grape'
1	'Carrot'	'Potato'	'Broccoli'	'Cabbage'
2	'Lettuce'	'Tomato'	'Celery'	'Radish'

If we use a row/column structure:

stock[1,2] = 'Tomato'

We can access single-dimensional arrays as we would a list; for multi-dimensional arrays we can use nested for loops.

In Python it is common practice to implement arrays by using lists of lists. In the example above we would create a list with three items, each of which is a list with four items.

## Dictionary

A **dictionary** is used to store data in key/value pairs, enabling the data to be searched by key values.

A dictionary:

- + is ordered
- + holds data in key/value pairs so the data can be referred to by the key
- + allows data items to be changed
- + does not allow duplicates.

### Worked example

Key	Data (test)
'Maths'	67
'English'	43
'History'	56

In Python we can create a dictionary structure – test – like this:

```
test = {'Maths':67,'English':43,'History':56}
```

We can ask the program to print the English test score using the key:

```
print(test['English'])
```

### Revision activity

In Python, create a program using the above list to print out each item in the list on a separate line using a for loop.

You can extend this task by getting a user input to select an index value in range so that a specified data value can be output.

### Exam tip

In the Worked example we have used row/column for the indices, but these tables are merely a representation of an array, and it is important to read the question carefully to see which notation is being used – do not assume row/column. If you are using a table to show an array, then it is always a good idea to indicate which notation you are using.

**Dictionary** A data structure using a key/value pair; the data are identified by the key.

### Revision activity

Use the list in the Worked example as a starting point and add a few more subjects. Write a program in Python that gets an input from the user for a subject and prints out the corresponding score.

## 2.1.4 Managing the variables within a program

REVISED

All non-trivial programs require variables. These are locations that can store data and can be accessed by the program. Each variable has a scope – the extent to which it can be seen within the different sections of the program.

### The use of local and global variables

**Global variables** are declared as part of the main program, and can be seen and used throughout the program and sub-programs. Because these variables are never destroyed, they can:

- + increase the program's memory requirements
- + cause errors in other parts of the program due to inadvertent changes being made to their value in another section of the code.

For programs written by teams of programmers, tracking these global variables is a major problem.

**Local variables** are declared and used exclusively within a sub-program.

- + Once the program exits the sub-program, the variables cease to exist.
- + The same variable name can be used in several sub-programs without any issues.
- + Teams creating sub-programs for a larger program do not need to keep track of local variable names used by other programmers.
- + It is possible for a local variable to have the same name as a global variable. The local version will be used whenever it is in scope without affecting the global variable of the same name.

It is considered good practice to avoid the use of global variables whenever possible.

### Naming conventions

When declaring a variable, it is good practice to use a meaningful name that identifies the purpose of the variable. This means it is easier to understand what a variable does when reading or modifying the code. The best code communicates what it does effectively without needing detailed references.

Choosing meaningful names will often require multiple words, but spaces are difficult for programming language interpreters to deal with. Words can be identified in a variable name using case styles.

- + Camelcase delimits the words by capitalising the first letter of second and subsequent words, as in the following example:

```
myFirstVariableName
```

Note that the first word is not capitalised.

- + Pascalcase is widely used in Java to name classes and interfaces. In Pascalcase all words have their first letter capitalised, as in the following example:

```
MyFirstVariableName
```

- + Snakecase, or snake\_case, is a very common method of naming variables. It uses only lower-case letters, and each word is separated using an underscore:

```
my _ first _ variable _ name
```

There are some key rules for naming variables:

- + Only alphanumeric characters and underscores (a–z, A–Z, 0–9 and \_) are allowed.

**Global variable** A variable declared in the main body of the program and visible to all parts of the program.

**Local variable** A variable declared within and used only within a sub-program; it is not visible outside that sub-program.

#### Key point

Java is the third most popular programming language, after Python and C. It is a general-purpose language used to build mobile and desktop applications, for games consoles, and applications for embedded systems up to big data processing.

- ✚ The name must start with a letter or underscore; it cannot start with a digit (0–9).
- ✚ The variable name is case sensitive – for instance, age, AGE and Age are different variables.
- ✚ It must not be a **reserved word** in the language, such as print or while.

## Now test yourself

TESTED

- State whether the following are integer, real, string, character or Boolean data types:
 

<b>a</b> 3.456	<b>c</b> 01@2d	<b>e</b> Thirty
<b>b</b> 12	<b>d</b> True	<b>f</b> B
- Describe two differences between a list and an array.
- What do we mean by the scope of a variable?
- What are the four rules for naming a variable?

**Reserved word** A word used by the language for a special purpose, which therefore cannot be used as a variable – for example, a command word such as print.

## 2.2 Operators

### 2.2.1 Mathematical operators in program code and algorithms

REVISED

There are several standard mathematical operators available in programming languages to enable a program to complete a range of calculations:

Operator	Example	Comment
Add	$a = x + y$	If $x = 6$ and $y = 2$ then $a = 8$
Subtract	$a = x - y$	If $x = 6$ and $y = 2$ then $a = 4$
Divide	$a = x / y$	If $x = 6$ and $y = 2$ then $a = 3$
Multiply	$a = x * y$	If $x = 6$ and $y = 2$ then $a = 12$
Integer division	$a = x // y$	If $x = 17$ and $y = 3$ then $a = 5$ It returns the whole number part after division
Modulus	$a = x \% y$	If $x = 17$ and $y = 3$ then $a = 2$ It returns the remainder after division
Exponent	$a = x ** y$	If $x = 2$ and $y = 3$ then $a = 8$ (i.e. $2^3$ or $2 * 2 * 2$ )

**Exam tip**

You may see some of these operators with different symbols. In the specification guidance and pseudocode, integer division is DIV, modulus MOD and exponent ^.

### 2.2.2 The purpose and use of relational operators

REVISED

Relational operators return True or False after comparing two values:

Operator	Pseudocode	Python	Example (Python)
Equals	=	==	If $x = 4$ and $y = 7$ , $x == y$ returns False If $x = 7$ and $y = 7$ , $x == y$ returns True
Not equal	<>	!=	If $x = 4$ and $y = 7$ , $x != y$ returns True If $x = 7$ and $y = 7$ , $x != y$ returns False
Less than	<	<	If $x = 4$ and $y = 7$ , $x < y$ returns True If $x = 7$ and $y = 7$ , $x < y$ returns False
Less than or equal to	<=	<=	If $x = 4$ and $y = 7$ , $x <= y$ returns True If $x = 7$ and $y = 7$ , $x <= y$ returns True

Operator	Pseudocode	Python	Example (Python)
Greater than	>	>	If x = 4 and y = 2, x > y returns True If x = 7 and y = 7, x > y returns False
Greater than or equal to	>=	>=	If x = 4 and y = 2, x >= y returns True If x = 7 and y = 7, x >= y returns True

### Worked example

A simple program shows how these relational operators work:

```
x = 5
y = 7
if y < x:
    print('y is less than x')
elif x < y:
    print('x is less than y')
else:
    print('x is the same as y')
```

This will output:

```
'x is less than y'
```

If we change the value of y to 3 the program would output:

```
'y is less than x'
```

If we change the value of y to 5 the program would output:

```
'x is the same as y'
```

## 2.2.3 Logical operators (NOT, AND, OR)

REVISED

To make more complex decisions we can use logical (or Boolean) operators.

- NOT inverts the value returned, so True becomes False and False becomes True.
- AND looks at two conditions and returns True only if both conditions are True.
- OR looks at two conditions and returns True if one or both conditions are True.

### Worked example

This example shows the result of using a number of logical and conditional operators:

```
x = 5
y = 7
z = 2
print(not(y < x))
print(y > x and x < z)
print(y > x or x < z)
```

### Output

True ← # y is not less than x so y < x is False, but 'not' inverts this to output True

False ← # y is greater than x so True, and x is not less than z so False. However, 'and' requires both conditions to be True so the output is False

True ← # 'or' requires only one of the conditions to be True so the output is True

## 2.3 File handling

Variables and data structures are temporary storage while a program is running. To store data more permanently text files can be used.

### 2.3.1 Using text files for input and output of data

REVISED

Before a text file can be used:

- + if the file does not already exist, it must be created
- + it must be opened for writing
- + data must be written to the file
- + it must be closed once the data have been written to the file.

Once the text file is created, it can be read by:

- + opening it for reading
- + reading the data
- + closing the file.

#### Worked example

In Python, if the file does not already exist, we need to create it before writing data to it. To do this we use the mode 'w+'.

This program creates a file called myfile.txt, writes a line of text to it, then reads it back and prints it.

```
x = open('myfile.txt','w+')
x.write('Testing 123')
x.close()
y = open('myfile.txt','r')
print(y.read())
y.close()
```

#### Revision activity

Write a program to open a file, add some data and then close it. Open the file again, output the data in the file and then add more data using the append 'a' option instead of 'w'. Open the file and output the data to show that it has been updated.

#### Now test yourself

TESTED

These questions use the specification pseudocode where appropriate to provide some practice.

- 1 If  $x = 4$  and  $y = 3$ , what is the value of  $a$  in the following pseudocode-based expressions?
 

<b>a</b> $a = x + y * 2$	<b>c</b> $a = 15 \text{ MOD } x$
<b>b</b> $a = (x * 6) / y$	<b>d</b> $a = 13 \text{ DIV } y$
- 2 If  $x = 3$ ,  $y = 4$  and  $z = 7$ , what is returned by the following pseudocode-based expressions?
 

<b>a</b> $x < y$	<b>c</b> $z = y$
<b>b</b> $x + y < > z$	<b>d</b> $x + y < = z$
- 3 Describe the process for writing data to a file that does not already exist.

#### Exam tip

The pseudocode used in this specification has the commands READ <File> <record> and WRITE <File> <record>. The meaning will be clear, but in order to gain marks, make sure you are familiar with the specification pseudocode.

## 2.4 Program structure

### 2.4.1 The use of sequence, selection and iteration within a program or algorithm

REVISED

In Section 1.2.3 we discussed the three major programming structures and how they can be implemented. See page 14 for examples of these three structures in pseudocode and in Python.

- 1 **Sequence** – an ordered list of instructions:
  - + The order is important if the desired outcome is to be achieved.
  - + No instruction is missed/skipped.
- 2 **Selection** (or branching) – the path through the program is determined by a condition:
  - + The condition returns True or False to determine which code to execute.
  - + Some code may not be executed.
- 3 **Iteration** – a section of code is repeated:
  - + for a fixed number of times
  - + until a condition is met
  - + while a condition is true.

**Typical mistake**

In Python almost everything is in lower case, but there is an exception: True and False must always start with a capital letter.

## 2.4.2 Write and debug code that makes use of sequence

REVISED

Determining the sequence of instructions affects the outcome of the program:

<code>num1 = 12</code>	<code>num1 = 12</code>
<code>num1 = num1 * 2</code>	<code>num1 = num1 + 2</code>
<code>num1 = num1 + 2</code>	<code>num1 = num1 * 2</code>
<code>print(num1)</code>	<code>print(num1)</code>
Output is 26	Output is 28

In these examples two lines of code are swapped over and produce different outputs. Given the similarity of the lines of code it might be difficult to spot such an error if it was part of a larger program; it would be necessary to trace the code to see the state of the variables in order to identify this error.

## 2.4.3 Write and debug code that makes use of selection (branching)

REVISED

Selection determines which program branch to follow based on the outcome of a condition. Multiple conditions are often required to determine which route to take through a program.

The if structure includes the option for multiple else if statements, i.e. other things to check if the first condition is false. A final else tells the program what to do if all conditions are false.

**Worked example**

A program to check if someone is old enough and tall enough to use a fairground ride:

```
age = int(input('Enter age '))
height = float(input('Enter height '))
if age <= 12:
    print('Too young to use this ride')
elif height <= 1.2:
    print('Too short to use this ride')
else:
    print('You can use this ride')
```

# MY REVISION NOTES

## T-LEVELS

THE NEXT LEVEL QUALIFICATION

# DIGITAL PRODUCTION, DESIGN AND DEVELOPMENT

Target success in Digital Production, Design and Development T Level with this proven formula for effective, structured revision. Key content is combined with exam-style questions, revision tasks and practical tips to create a revision guide that you can rely on to review, strengthen and test your knowledge.

With My Revision Notes you can:

Plan and manage a successful revision programme using the **topic-by-topic planner**.

Enjoy an interactive approach to revision, with clear topic summaries that consolidate knowledge and **revision activities** that put the content into context.

Build, practise and enhance your exam skills with **Exam tips** and **Now test yourself** questions.

2.8.4 How to apply root cause analysis to solve problems

Root cause analysis is a systematic way of finding and identifying the root cause (or causes) of a problem.

- The 'five whys' technique is a problem-solving approach that relies on asking 'why?' five times to identify the root cause of a problem.
- Each time you ask why a problem occurred, the answer then becomes the premise of the next question. By doing this, you dig deeper and deeper into the true cause of the issue.
- Root cause analysis is used when issues occur that hinder the effective use of the system. For example:
  - loss of service
  - loss of data
  - poor system performance
  - downtime or system crash.
- If a problem occurs, then it is important to:
  - understand and define the problem
  - collect data and information related to the problem
  - identify the cause, or causes, of the problem (do not assume it will be just one cause)
  - identify which cause to deal with first and a process to deal with all causes
  - identify and implement the necessary solutions.
- Once a solution has been identified and implemented it is important to test the modifications and monitor the system for any consequential problems.

2.8.5 How to construct an effective test plan

The purpose of testing is to make sure the program:

- functions as expected under normal circumstances
- functions under extreme conditions
- is robust and does not break easily.

Testing should be destructive and seek to make the program fail over so that we can be satisfied that it works.

The test plan should include the test data, the reason for the test, the expected outcome and the actual outcome:

Test data	Type of test data	Reason for test	Expected outcome	Actual outcome
This must be actual data, not a description of the type of data	This should indicate what type of data is normal, boundary, extreme, etc.	The reason for the test and what it will show/check	What is expected to happen, any output, or that the data are accepted/rejected	The actual result of the test

- Valid data (or normal test data) are data that are within the normal range and of the correct type expected during normal use. This should be accepted without causing any errors and should produce the expected outcome.
- Valid extreme data are normal data that are at the extreme end of the range for acceptable data and should be accepted without causing any errors and output the expected results. This type of test data checks that the boundary values are set correctly.

Invalid data are data that are of the correct type but outside the expected range and should be rejected, without causing any errors.

Invalid extreme data are data that are outside the range but just on the limits of unacceptable, like the valid extreme data test, this tests the boundary conditions. These data should be rejected by the program without causing any errors.

Erroneous data are data of the wrong type, for example string instead of integer, and should be rejected by the program without causing any errors.

Typical mistakes

- It is a common for people to describe the test data used or to provide a range of values. That is not sufficient, the actual values must be stated.

Now test yourself

1. What do we mean by software testing?
2. Why is it important to identify the hardware requirements for a system?
3. Describe three elements of system testing.
4. What is meant by root cause analysis and when is it used?
5. What is required in a test plan?

Summary

In this chapter you learned about:

- Program data
- Data types
- Declaring and using appropriate data types
- Data structures
- Managing variables within a program
- Operators
- File handling
- Use of test files for input and output of data
- Program structure
- The use of sequence, selection and iteration within a program or algorithm
- Writing and debugging code that makes use of sequence
- Writing and debugging code that makes use of selection
- Writing and debugging code that makes use of iteration
- Declaring and calling functions and procedures
- Standard searching and sorting algorithms
- Built-in functions
- Benefits and drawbacks of using pre-written Python
- Selecting and justifying the use of pre-written Python
- Validation and error handling
- The need for input validation
- The need for reliable and robust code
- Maintainable code
- The accepted style conventions, and how they are implemented to make code more readable and maintainable
- Testing
- The importance of testing for all components
- The use of testing and quality assurance methodologies to identify problems and bugs
- How automated and functional testing tools can be applied to test digital systems and code
- How to apply root cause analysis to solve problems
- How to construct an effective test plan

Exam practice

1. A program uses local and global variables.
  - a. Explain what is meant by a global variable. (2)
  - b. Describe two advantages of using a local variable. (4)
2. State which of the following are valid variable names in Python.
  - a. my\_name
  - b. 01\_numbers
  - c. \_your\_name
  - d. yourName
  - e. myName23
3. There are different types of variable.
  - a. Explain what is meant by casting a variable as an integer. (2)
  - b. Describe one use for the data type Boolean. (2)
4. List is one of the three most commonly used data structures.
  - a. Identify the six major characteristics of a list data structure. (6)
  - b. If myList = ['tree', 'flower', 'fruit', 'vegetable', 'plant'], what is the value of myList? (1)
  - c. What is the index value for the item 'plant' in the list? (1)
5. a. Describe two features used when writing programs that make them easier to follow. (4)
  - b. Describe what is meant by a function in programming. (2)
  - c. Describe the use of parameters with functions and procedures. (2)

Improve exam technique through **exam-style questions** and sample answers with commentary from our expert authors.

**Boost**

This title is also available as an **eBook** with **learning support**.

Visit [hoddereducation.com/boost](https://www.hoddereducation.com/boost) to find out more.

**HODDER Education**  
+44 (0)1235 827827  
[education@hachette.co.uk](mailto:education@hachette.co.uk)  
[www.hoddereducation.com](https://www.hoddereducation.com)

Schools have a **Licence to Copy** one chapter or 5% for teaching

**CLA** Copyright Licensing Agency

ISBN 978-1-3983-8450-7

