In pseudocode
Here is that structure shown as pseudocode:

```
Count ← 0
CarryOn ← "Y"
WHILE CarryOn = "Y"
    Count ← Count + 1
    PRINT "Type Y to continue"
    READ CarryOn
ENDWHILE
PRINT Count
```

Which line of the pseudocode increases Count by 1? Which row lets you input "Y" to carry on?

Flowchart
In the right margin you can see the same algorithm shown as a flowchart.

In Python
Here is the completed Python program.

```
# WORKED EXAMPLE
# Count how many
print("Begin count")
Count = 0
CarryOn = "Y"

while CarryOn == "Y":
    Count = Count + 1
    print("Count = Count + 1")
    CarryOn = input("Do you want to continue (Y/N)?")

print("Count ended")
print("you counted :", count)
```

Make this program for yourself and check that it works.

Learning activity
Using the example on this page create a Python program for at least one of the following. If you are confident make sure you do activity 3: Stretch and challenge.

1. Write a program for an official at a marathon to use to count the runners as they cross the finishing line.
2. Write a program for a farmer so he can count how many new lambs are born in his flock.
3. Stretch and challenge: Write a program to control a car park barrier. It must count how many cars go into the car park. In this example the loop will stop if the number in the car park reaches 100. At this point, no more cars can enter.
4. Explain why we use a condition-controlled loop and not a counter-controlled loop in this program.

Test yourself
1. What are the names of the two variables used in the Python program?
2. What are the data types of the two variables?
3. What is the exit condition of the loop? In other words, what makes the loop stop?
4. Explain why we use a condition-controlled loop and not a counter-controlled loop in this program.
Solution development

Syllabus reference

2.2.1 Programming concepts
Learner should be able to:
understand and use the concept of totalling (work out the purpose of a given algorithm; explain standard methods of solution; produce an algorithm for a given problem (either in the form of pseudocode or flowchart)).

2.1.2 Pseudocode
Learners should be able to understand totalling (for example: Sum ← Sum + Number).

Calculate sum total

Introduction
In the previous section you learned a standard algorithm and code to count how many times something happened. It used a loop. In this section you will learn how to create a program that calculates a total by adding values together.

Sum total
In this section you will create a program to add up a series of values. This is a very useful basic program structure. You can adapt the program structure for many different needs. For example, you could use it to add up the bill in a shop or restaurant; add up the total score a student gets in an exam; or add up the total production of a factory.

In everyday speech we use the word “sum” to refer to any calculation, for example: “I am good at doing sums”. In mathematics “sum” is used to mean the total you get by adding up a list of numbers. We will use the word in that sense.

Which type of loop to use
Should you use a condition-controlled loop or a counter-controlled loop to calculate a sum total? It depends on the following:

● Sometimes you know in advance exactly how many values you will need to add together. For example, if you are adding up the marks a student has gained for the questions in an exam, you know exactly how many questions there are, so you know how many items (values) need to be added together. In this case, you would use a counter-controlled loop.

● Sometimes you do not know how many values you need to add together. For example, if you are adding up customers’ bills in a shop, every customer could buy a different number of items. You would not know in advance how many values you would need to add together, so you would use a condition-controlled loop.

The worked example in this section uses a condition-controlled loop.

Program structure
Here is the basic structure of a program to add up the sum total of a series of numbers:

● Create a variable called sum. Set its value to 0.

● Make a loop (it can be a condition-controlled or counter-controlled loop).

● Every time the loop increments, the user enters a value.

● Every time the loop increments, the input value is added to the variable sum.

● When the loop stops, the value of sum is printed out.

In pseudocode
Here is the sum total algorithm shown in pseudocode. This example uses a condition-controlled loop:

```plaintext
Sum ← 0
CarryOn ← "Y"
WHILE CarryOn = "Y"
   READ Number
   Sum ← Sum + Number
   PRINT "Type Y to continue"
   READ CarryOn
ENDWHILE
PRINT Sum
```

Flowchart
In the right margin you can see the same algorithm shown as a flowchart. The flowchart loop will increment until the user enters any value except “Y”.

In Python
Here is the completed Python program.

```python
# WORKED EXAMPLE
# Sum (Add together)
print("begin to add up the total")
Sum = 0
CarryOn = "Y"
while CarryOn == "Y":
   Number = input("enter the next value to add: ")
   Number = int(Number)
   Sum = Sum + Number
   CarryOn = input("Do you want to continue (Y/N)?")
print("Adding up completed")
print("The total was ", Sum)
```

Enter this program yourself and check that it works.

Learning activity
Create Python programs for at least one of the following. These activities increase in difficulty from the first to the fourth. Make sure you do at least the first one.

1. Make a program for a shopkeeper that will enter the cost of each purchase and display the total customer bill.
2. Adapt the shop program so that the user can enter and add up decimal number values.
3. Make a program for a teacher to add up the marks from ten questions to give a total test mark. It must be counter-controlled.
4. Make a program for a mathematician. The mathematician will enter a series of numbers. The computer will calculate the product.

Test yourself
1. Explain the reason for this line in the Python program:
   Number = int(Number)
   2. Create a pseudocode algorithm using a counter-controlled loop to add up exactly ten values.
   3. You have learned that the sum of a series of numbers is calculated by carrying out a series of additions. If instead you use a series of multiplications, the result is called a product. Create a flowchart that calculates the product of a series of numbers.
Calculate an average

Introduction
In the last two sections you learned how to use program methods to count up and to add up. By combining these two methods you can create a program that works out an average. This is a good way to practise your programming skills.

Average (mean)
This program will calculate an average. The mathematical term for an average is a “mean”.

The mean value is calculated as follows.
1. Count how many values there are (the count).
2. Add up the total of the values (the sum).
3. Divide the sum by the count to give the average.

Which type of loop to use
Should you use a condition-controlled loop or a counter-controlled loop for this program? Once again, it depends whether you know the number of values in advance. The worked example in this section uses a counter-controlled loop.

Program structure
You have seen two examples using a condition-controlled loop. In this section for a change we will use a counter-controlled loop. Here is the program structure:
● Create a variable called Sum with the value 0.
● Create a variable called Count, which stores a number input by the user.
● Start a counter-controlled loop: the start value is 0, the stop value is Count.
● Every time the loop increments, the user enters a value. The input value is added to the variable Sum.
● When the loop stops, Sum is divided by Count to give the average.

In pseudocode
Here is the structure shown in pseudocode. It uses a counter-controlled loop:

```
Sum ← 0
PRINT "How many values do you want to enter?"
READ Count
FOR i ← 1 TO Count
    READ Number
    Sum ← Sum + Number
NEXT i
Average ← Sum / Count
PRINT Average
```

Flowchart
In the right margin you will see the same algorithm shown as a flowchart. This shows what a counter-controlled loop looks like in a flowchart.

In Python
Here is the completed Python program.

```
# WORKED EXAMPLE
# Mean (Average)
Count = int(input("How many values will you enter? ")
Sum = 0
for i in range(0, Count):
    Number = int(input("Enter a value ")
    Sum = Sum + Number
print("Adding up completed")
Mean = Sum/Count
print("The average was ", Mean)
```

Enter this program for yourself and check that it works.

Test yourself
1. Which line in the pseudocode starts the counter-controlled loop?
2. What is the name of the counter variable in all these examples?
3. What variable sets the upper range of the counter loop?
4. Which boxes in the flowchart are inside the counter loop?
5. List all the variables in the Python program, and explain what they are used for.

Learning activity
Create Python programs for at least one of the following. Make sure you do at least the first one.
1. Make a program for a scientist that adds up the weight of six eggs in grams, and gives the average weight. A typical egg weighs 50–75 grams. Use integer values.
2. A marathon has 15 competitors. Make a program that records the number of minutes each runner took to complete the race and calculates the average time taken. A typical time would be 250 minutes. Use floating point values.
3. Adapt the marathon program to use a condition-controlled loop. You can use this program to work out an average when you don’t know how many runners will complete the race.
Solution development

**Verification**

**Introduction**
In this section you will learn to use programming methods to check input. You will also see why it is important to include error checks in a program.

**Error checks**
A program will go wrong if the user enters the wrong data. If the input is wrong then the output of the program will definitely be wrong. Sometimes wrong data can cause the program to crash, or get stuck in an endless loop.

For this reason many programs include input checks. User input is checked before it is used in the program. Typically, the procedure is as follows.
1. Input from the user is stored in a temporary variable.
2. The content of the temporary variable is checked.
3. If the input passes the error check, then the value is used in the program.

**Error message**
Typically, if the input is wrong there is an error message. This tells the user what he or she has done wrong. Sometimes the user is given a chance to enter the data again.

**Types of check**
There are two types of check:
- verification
- validation.

Learn about verification in this section. Learn about validation on pages 228–229.

**Verification**
Verification means entering the data twice. The two attempts will be stored as two different variables. The computer will check that the two variables match. If they do not match then there has been a mistake in data entry.

A verification check is often used when you change your password on the computer. You have to enter the new password twice. That is because it is very important that you don’t make a mistake that would lock you out of the computer.

**Use selection or repetition**
You can use either selection (if... else...) or repetition (while loop) for verification.

**Selection**
One way to verify an input is to use an if... else structure, as follows.
1. The user enters a value and it is assigned to a variable.
2. The user enters the value again, and it is assigned to a second variable.
3. An if statement is used to compare the two variables.
   - If the two variables match, the input has passed the verification check, so you can use it in the rest of the program.
   - If the two variables do not match, else will show an error message.

The disadvantage of this method is that the user only gets one chance to enter the right data.

**Repetition**
An alternative method is to use a condition-controlled loop. It will loop until the input is correct.
1. The same data is entered twice and stored in two variables.
2. A while loop is used. It will loop if the two values do not match.
3. Inside the loop the user is given a chance to re-enter the data.
4. When the two values match, the loop will stop.

The advantage of this method is that it gives the user several chances to get the value right.

**Example**
Here is an example of a Python program that uses if... else for password verification.

```python
# WORKED EXAMPLE
# Verification
Try1 = input("Enter new password : ")
Try2 = input("Enter password again: ")
if Try1 == Try2:
    password = Try1
    print("Password has been reset")
else:
    print("The two passwords do not match")
```

Make this program yourself and check that it works.

**Repetition: Use a while loop**
Here is an example of a Python program using a while loop for verification.

```python
# WORKED EXAMPLE
# Verification with loop
Try1 = input("Enter new password : ")
Try2 = input("Enter password again: ")
while Try1 != Try2:
    print("The two passwords do not match")
    Try1 = input("Enter new password : ")
    Try2 = input("Enter password again: ")
Password = Try1
print("Password has been reset")
```

Enter this program yourself and check that it works.

Test yourself
1. List the three variables used in these Python programs. What data type is each one?
2. Explain the purpose of each of the three variables in the program.
3. Which of the two programs is the most useful for password input? Explain your answer.

Learning activity
Create pseudocode and flowchart algorithms to match the two Python programs shown on this page.
Validation

Introduction
You have learned why error checks are important. Many programs include error checks. In this section you will learn about a type of error check called validation.

Validation
Validation means checking that data is valid. That means the input data is sensible in context. In general, when making a program we know what sort of input data to expect. Data that is not what we expect can crash the program or produce incorrect results. Validation blocks this data.

Validation criteria
Validation criteria are rules about what data can be accepted by the program. When data is entered it is checked against the validation criteria. If data does not meet the criteria it is not used in the program.

Selection or repetition?
In this section you will look at examples of validation in programming. As with verification, we can use either selection or repetition. The advantage of using repetition is that it give the user many chances to enter the right data.

Validation checks
Validation checks data using criteria. There are many different types of validation criteria:

- Range checks are only used with numerical variables. The number must be within a valid range. For example, a teacher may need to enter the age of a student at a secondary school where the age could not be below 12 or above 18 years.
- Length checks are used mainly with text variables, particularly code numbers. The right number of characters must be entered. For example, a mobile phone number may need to have exactly 11 digits.
- Type checks can be used with all variables. You have learned that every variable must have a data type. A type check makes sure that input is of the right data type. For example, if you are entering an age it must be a number value; if you are entering a name it must be made of letters of the alphabet and so on.
- Check digits (in 2.1 Data transmission you learned about the use of check digits): a check digit is calculated from a long number using a mathematical formula. The check digit is entered along with the number. The computer uses the same formula to calculate the check digit. The two check digits should match. Otherwise, there has been a mistake in data entry.

Examples
There are many different ways of making validation checks in Python and in other programming languages. We will look at some examples.

This Python program has a range check. The user inputs how much money she wants to take out of her bank account. If the value is below 0, the number is rejected.

```python
# WORKED EXAMPLE
# Range check
balance = 1000.00
money = input("How much money do you want from your bank account: ")
money = float(money)
if money < 0:
    print("You cannot withdraw less than 0")
elif money > balance:
    print("You do not have that much money")
elif money > 250.00:
    print("You cannot withdraw more than $250.00 a day")
else:
    print(money, " has been withdrawn from your account")
    balance = balance - money
print(balance, " remains in your account.")
```

Here is the same program with extra range checks built in. This program uses elif.

```python
# WORKED EXAMPLE
# Range check
balance = 1000.00
money = input("How much money do you want from your bank account: ")
money = float(money)
if money < 0:
    print("You cannot withdraw less than 0")
elif money > balance:
    print("You do not have that much money")
elif money > 250.00:
    print("You cannot withdraw more than $250.00 a day")
elif money > 50.00:
    print("You cannot withdraw more than $50.00 a day")
else:
    print(money, " has been withdrawn from your account")
    balance = balance - money
print(balance, " remains in your account.")
```

Finally, this program uses a loop to repeat the input until the data is in the right range.

```python
# WORKED EXAMPLE
# Range check
age = input("How old are you: ")
age = int(age)
while age < 0:
    print("You cannot be less than 0")
age = input("How old are you: ")
age = int(age)
print("Age data accepted")
```

Test yourself
1. What variables are used in the first program and what are their data types?
2. Explain the range checks used in program above that uses elif.
3. What is the error message in the third program (the one with the loop)?
4. Describe the four types of validation check you have learned.
5. Explain the difference between validation and verification.

Learning activities
1. Enter the Python programs shown on this page and try them out to see what happens.
2. Create a program for a bank cash machine. It has two loops in it. Base the program on the following:
   - Include a variable called Balance with a starting value of 1000.
   - The user inputs a four-digit password. The program loops until the user enters the number 9080.
   - The user inputs the amount he or she wants to withdraw from their account. The program loops until this is smaller than Balance.

See also:
2.1 Data transmission (check digits)
8.3 Selection (Python `elif`).

Syllabus reference
2.1.1 The need for validation
Learners should be able to understand the need for validation (which could include range checks, length checks, type checks and check digits); explain standard methods of solution; comment on the effectiveness of a given solution.
Testing and evaluation

Test data

Introduction

Testing a program means trying it out. A programmer will always test a program to make sure it works properly. If there are problems or errors they will be fixed. In this section you will learn about how to design good tests with suitable test data.

Input and output

A completed program that runs on a computer is an example of software. You have learned that software is made for clients or customers. Software has input and output:

- Input is the data the user enters into the computer.
- Output is the result the computer gives to the user.

The client needs the output of the software. That is why people buy software. Programmers must know what the output should be. They must make software that produces the right output.

Testing

Before software is passed to the client it must be tested. To test software:

- Try out different inputs to the software.
- Check what output you get.

The output you get should match the output the client wants:

- If the output matches what the client wants, the software has passed the test.
- If the output does not match the client’s requirements, the software has failed the test. The programmer must fix the software.

Programmers will plan a whole series of tests. They will record every test and what the results are. They will analyse test results to make sure the software is working properly.

Test data

To test software you must input data. That is called the test data. To carry out a full range of tests you must try many different types of test data.

As a rule, you should try three different types of test data:

- Normal data – this is the normal input that you would expect users to enter when they use the software in real life. You must make sure that in normal use the software works as expected and accepts the input.
- Extreme data – test the limits of your software by entering large or small numbers that are at the upper and lower limits of the acceptable range. The program should accept these inputs.
- Abnormal data – this is data that should not be entered, for example when a user enters letters instead of numbers, or presses the Enter key at the wrong time. We use these tests because users sometimes make mistakes and we must check what happens when they do. The best result is for the data to be rejected and an error message to be shown.

Example

A programmer made a program to add up the bill in a coffee shop. The user enters the cost of drinks and the software shows the total amount.

The program was tested. Here are some of the values that were input as test data:

- the prices of two ordinary cups of coffee
- a very big order with 100 different coffees
- the number 1,000,000
- letters of the alphabet instead of numbers.

Can you work out which of these are normal, extreme and abnormal data?

Test yourself

1. What is test data: the input, the program code or the output?
2. Explain why programmers test a program before they pass it to the client.
3. An input prompt says “enter a four digit number”. The test data was the letter “X”. Explain why.
4. An input prompt was enter your age. The test data was 999. Explain why.
5. A test produced the expected results. What does that tell the programmer?
6. A test produced output that was different from what the client wanted. What must the programmer do?

Learning activity

A voice recognition program records samples of human speech and shows the words as text on the screen.

1. What are the inputs and outputs of this software?
2. Design some tests for this program. Say what test data you would use, and what results you expect.
Solution development

Effective

A good program is effective, which means that the program produces the right effect. It produces the right outputs. Programmers need to decide whether each program they have made is effective.

The programmer needs to know what outputs the client wants. The outputs are the client's requirements. The program is effective if it:

- produces all the client's requirements
- produces outputs with no errors in them
- works in a way the client likes.

If the program is effective the programmer can give it to the client. If the program is not effective the programmer must make changes and improvements.

Test effectiveness

After the program is completed it is tested. Testing checks the outputs of the program. The outputs of the program are compared with the client's requirements.

Examples

The client wanted a program that would input five numbers and output the largest number. A programmer made this program. Then the programmer had to test the program.

Test plan

The programmer planned tests. Each of the client's requirement had to be tested with normal, extreme and abnormal data.

The test plan is set out in a table. For each test we need to know the purpose of the test, the test data and the expected output. The expected output is what we would see if the program worked without any problems.

Here is an example test plan. This plan shows three tests. In real life the programmer would do more tests than this.

Evaluation

Introduction

You have learned that software is made for clients. The client needs the output of the software. The programmer must make software that produces the right output. In this section you will learn how to evaluate effectiveness by looking at the outputs.

Record test results

Test results are recorded in the table. They are recorded in the "Actual output" column.

Here is an example. The first test produced the expected outcome. The next two tests did not produce the expected result.

Analyse test results

The programmer compared the actual results to the expected outcomes. If the actual and expected outputs are the same, no action is needed. If the actual and expected outcomes are different, further action is needed.

The results of this analysis are shown in the final column of the test table.

Report on effectiveness

Finally, you can sum up the effectiveness of the solution. You will summarise the findings of all the tests. You will explain whether the solution works. Is the program effective? Does it produce the results the client wants? Does it always produce the correct outputs? Is more work needed?
Solution development

Test an algorithm

You have learned that algorithms are used to set out the logic of a program. An algorithm is made before the programmer starts to write code. It is a good idea to test the algorithm before you make the program.

This section explains how you can test an algorithm. An algorithm is not working software. You cannot run it and see what happens. Instead you use a trace table. A trace table records the values of each variable in an algorithm.

1. The table has a column for each variable in the algorithm.
2. The table has a row for every line of the algorithm.
3. You go through an algorithm line by line.
4. You record the value of each variable at every line.

Test yourself

1. The trace table has a column for each...
2. What should be entered into each row of a trace table?
3. Explain two ways that a value can be assigned to a variable.
4. A trace table normally counts down line by line – but not always. Why not?
Solution development

Trace tables (loops)

Introduction
In the previous section you created a short trace table. In this section we will create a trace table for an algorithm with a loop in it.

Example
Here is the algorithm we will test. The lines are numbered to help us make the trace table:

1. Largest ← 0
2. CarryOn ← "Y"
3. WHILE CarryOn = "Y" DO
4.    READ Number
5.    IF Number > Largest THEN Largest ← Number ENDIF
6.    PRINT "Type Y to continue"
7.    READ CarryOn
8. ENDWHILE
9. PRINT Largest

What does this algorithm do?

Test data
We will test the algorithm by entering the following test data:

Number = 10
CarryOn = "Y"
Number = 5
CarryOn = "N"

We will see what output we get.

Shortening the trace table
The trace table could get very long. To make the trace table shorter we will only include lines where a value is assigned to a variable.

Trace table
Here are the first few lines of the trace table. In the first two lines, values are assigned to the variables Largest and CarryOn. In Line 4 input is entered and assigned to the variable Number.

<table>
<thead>
<tr>
<th>Line</th>
<th>Largest</th>
<th>CarryOn</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>Y</td>
<td>10</td>
</tr>
</tbody>
</table>

Logical test
Line 5 has a complete IF statement on one line:

5. IF Number > Largest THEN Largest ← Number ENDIF

The IF statement includes a logical test. The test is Number > Largest. Check the values in the trace table:

Number = 10
Largest = 0
Number is bigger than Largest. Therefore, the command is carried out:
Largest ← Number

The value of the variable Number is assigned to the variable Largest.

Loop
Now the algorithm goes around a loop. Let us trace this carefully line by line:

● Line 6 is an output. There are no changes to variables.
● In line 7 the test data Y is entered into the CarryOn variable. This does not change the value of the variable.
● Line 8 marks the end of the loop. We go back to the top of the loop. We go back to line 3.
● Line 3 is a logical test. The test result is True, so the loop continues.
● Lines 4-7 are carried out again, using the next lot of test data.
Here is the completed test table, following the loop.
In line 7 the value N is entered. We go back to the top of the loop. Line 3 is:

3. WHILE CarryOn = "Y" DO

This test result is False. The loop stops. The algorithm goes to line 9. The variable Largest is printed out. It has the value 10.

Test yourself
1. What does the algorithm on this page do?
2. Create a trace table for the algorithm shown on this page. Use the test data below. What is the output?

Number = 10, Y, 20, Y, 99, Y, 55, N

Learning activity
In 9.1 Worked examples you created a series of different algorithms:

● count
● total
● average (mean)
● verification
● validation.

In 9.2 Testing and evaluation, you learned how to choose suitable test data.
Your challenge in this section is to test the algorithms you made.
1. Pick one of the algorithms you made in 9.1. Write out the algorithm. Number every line.
2. Choose suitable normal test data for this algorithm.
3. Produce a trace table showing the value of every variable at every line of the algorithm.
If you are confident, produce trace tables for all the algorithms you have made.
Trace tables (flowcharts)

Introduction
You have created trace tables for algorithms made in pseudocode. In this section you will learn to create a trace table for a flowchart algorithm.

Test data
We will use this test data:

<table>
<thead>
<tr>
<th>Number</th>
<th>Continue</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Y</td>
</tr>
<tr>
<td>8</td>
<td>N</td>
</tr>
</tbody>
</table>

Make the trace table
There are three variables in this algorithm. They are called Sum, Number and Continue. We make a trace table with a column for each of these variables.

The trace table has a row for each box in the flowchart:

- In box 1: Sum is given the value 0.
- In box 2: data is input to the variable Number. The test data is 7.
- In box 3: the values of Sum and Number are added together and assigned to the variable Sum.
- In box 4: data is input to the variable Continue. The test data is Y.

The trace table shows these changes.

<table>
<thead>
<tr>
<th>Line</th>
<th>Sum</th>
<th>Number</th>
<th>Continue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>7</td>
<td>Y</td>
</tr>
</tbody>
</table>

Finish
Box 5 is a decision box. Like every decision box it shows a test. The test is:

Continue <> "Y"

That means "The value in the variable Continue is different from Y". This test result is False, so we take the NO arrow out of the decision box. The NO arrow goes down. We have gone out of the loop. The final box is box 6. The variable Sum is output. It has the value 15. That is the result of the algorithm.

Loop
Box 5 is a decision box. Like every decision box it shows a test. The test is:

Continue <> "Y"

That means "The value in the variable Continue is different from Y". This test result is False, so we take the NO arrow out of the decision box. The NO arrow goes off to the right of the decision box. It loops back up the algorithm. We end up above box 2.

Test yourself
1. What calculation is carried out by the algorithm on this page?
2. Suggest some different test data you could use to test this algorithm.
3. How does numbering the flowchart boxes help us to make the trace table?
4. How do we decide how many columns there should be in the trace table?

Learning activity
You have created flowcharts for these other algorithms:
- count
- average (mean)
- verification
- validation.

Create a trace table for at least one of these flowcharts. Choose suitable test data.
**Introduction**
You have made algorithms using pseudocode and flowcharts. You have used trace tables to test the algorithms. In this section you will put all your skills together to help you to analyse and understand any algorithm.

**Purpose of an algorithm**
You have learned that an algorithm takes data and processes it to produce information. For example the flowchart on page 223 took a series of numbers as input. The output was the sum (total) of the numbers.

The purpose of an algorithm is to turn input data into output information. To describe the purpose of an algorithm you must describe the relationship between input and output values.

In this section you will learn some methods for understanding the purpose of an algorithm, including:
- spotting familiar algorithms
- reflecting on variable names
- noticing what calculations are used
- noticing the structure of the algorithm
- using a trace table if you still cannot work it out.

**Familiar algorithms**
You may recognise a familiar algorithm from your learning activities. Look back at 9.1 Worked examples. Learning to recognise these common algorithms will make you a better programmer.

**Naming conventions**
A well-designed program or algorithm will have sensible variable names. The name of the variable will tell you its purpose and help you to understand the algorithm. For example, on page 236 the variable is called `Largest`. On page 223 the variable is called `Sum`. These names help you to understand what each algorithm does.

However, variable names do not always give you all the information you need. For this reason, you need to consider other elements.

**Calculations**
Algorithms nearly always include calculations. In pseudocode, calculations use the arrow symbol to assign a calculated value to a variable:

```
Sum ← Sum + Number
```

Flowcharts use a rectangular box. The calculation is shown inside the box.

Calculations are a key way that data is transformed into information. Look at the calculations in the algorithm. Which arithmetic variables are used?

Describe in words what you see. This will help you to understand the purpose of the algorithm.

**Structure**
Look at the structure of the algorithm. There are two important structural features to look at:
- Are there any IF structures?
- Are there any loops?

Describe in words the structure of the algorithm. This will help you understand what it does.

**IF structures**
Look at the IF structure of the algorithm. Answer the following questions.
1. What is the logical test?
2. What actions are carried out if the test result is True?
3. What actions are carried out if the test result is False?

**Loop structures**
Look at the loop in the algorithm. Answer the following questions.
1. Is it a counter-controlled or a condition-controlled loop?
2. What actions are carried out inside the loop?
3. What is the exit condition of the loop?

**Trace tables**
It can take some time to make a trace table for an algorithm. You need to choose suitable test data. You may need to do several trace tables, using different test data.

Try the simpler analysis methods first, but if you are still stuck use a trace table. Thorough testing using a trace table will help you to understand any algorithm.

---

**Test yourself**
1. How will naming conventions help you understand an algorithm?
2. What symbol is used in a pseudocode calculation command?
3. What flowchart symbol is used for the exit condition of a loop?
4. Explain two ways in which logical tests are used in algorithms.
Find errors in algorithms

Introduction
You have learned how to test algorithms and work out the purpose of algorithms. In all the examples so far the algorithms have been well designed and free of errors. In this section you will learn how to work with algorithms that have errors. You will find out how to identify and correct the errors.

Types of error
There are three types of error you can find in an algorithm or computer program:
- syntax errors
- run-time errors
- logical errors.

Syntax errors
“Syntax” means the rules of a language. This includes the rules of pseudocode. The lines of pseudocode must exactly match the rules of the language. Syntax errors occur when the programmer has not entered the commands in the correct way. For example, the first line of a pseudocode counter loop is written like this:

```
WHILE CarryOn = "Y" DO
```

If you omitted the word DO, this would be a syntax error.

Correcting syntax errors
When you write program code, the compiler or interpreter will tell you if it spots a syntax error. The program will halt. When you write in pseudocode you need to spot your own errors. You have to learn the rules of the language and remember them. It is usually easy to correct a syntax error. You retype the line to match the rules of the language.

Run-time errors
Run-time errors do not break the rules of the language. The program code will run, but when you run the program, it will go wrong. Common run-time errors are:
- endless loop
- divide by 0.

Endless loop
Every loop must have an exit condition. The exit condition is a logical test. It tests the value of a variable. If the test result is True the loop will stop. There must be a way to stop the loop. There must be a command inside the loop that changes the variable.

If you cannot make the variable match the exit condition, the loop will never stop. To correct it you must add a command inside the loop that lets you stop the loop.

Divide by 0
It is mathematically impossible to divide by 0. If you try to do this, your program will crash. This is a danger if you have a calculation using division in your program. Use validation to prevent the user from entering a 0 in this case.

Logical errors
A program without syntax or run-time errors can still have logical errors. A logical error means the program works properly, but it does not do the right thing. For example, a programmer wrote a program to add a service charge to a bill. He used the minus sign instead of the plus sign. The program worked, but it gave the wrong result.

Find logical errors
To find logical errors you must:
- have a clear statement from the client about what the program is supposed to do
- carry out a wide range of tests
- compare the results of the tests with what the client wants.

If there are logical errors, the test results will not show the right values. You must change the program to make it do what the client wants.

Syllabus reference
2.1.1 Identify and fix errors in algorithms
Learners should be able to identify errors in given algorithms and suggest ways of removing these errors.
See also:
Chapter 8 Programming
5.2 Testing and evaluation

Find three errors in this pseudocode algorithm. The lines are numbered to make it easier to discuss the structure:

```
1. Product ← 1
2. Continue = "Y"
3. WHILE Continue = "Y" DO
4.    READ Number
5.    Product ← Product * Number
6. ENDIF
7. PRINT Product
```

1. Answer these questions.
   a. Identify the line or lines where each error occurs.
   b. Say whether each error is a syntax, run-time or logical error.
   c. Explain the error.
2. Write a corrected version of the algorithm and explain what it does.

1. What is a syntax error?
2. What command must you find inside a condition-controlled loop?
3. What do you call an error that produces output that does not match the client’s needs?
4. Explain how testing will help you spot errors in an algorithm.
Solution development

Understand the requirement
An algorithm processes inputs to create outputs. To plan an algorithm you must:
- be clear about what outputs are required
- work backwards from the required outputs
- know what processing is needed to make the output.
Make sure you have a clear statement of the purpose of the algorithm before you start work. This needs to include:
- what values will be input
- what calculations and other processes will occur
- what output is required.

Re-use and adapt
If you are lucky you will already know an algorithm that matches the requirement. Good programmers will know many different algorithms. They will know algorithms they have made themselves. They will know the work of other programmers. This makes their job much easier and quicker.
In other cases you might know an algorithm that is almost right for the purpose. You may have to adapt the algorithm slightly, but this is still much easier than making a new algorithm.
For example, a programmer made an algorithm to calculate the charge for a meal in a restaurant. Later, she adapted it to work out the entrance fee for a cinema. With a few changes the algorithm worked, saving the programmer a lot of work.

Create an algorithm
Sometimes you will need to create an algorithm starting from nothing. Here are some tips.

Inputs, outputs and variables
Think about the client's requirements. Think about all the inputs and outputs you need, then:
- add lines to enter input and store it as a variable
- add an output statement at the end of the algorithm to show the result.

Calculation
Consider what calculations will be required to turn inputs into outputs. Think about what arithmetic operators you will need; then:
- between the input and output lines, add any calculations that are needed.

Selection
Selection means the use of IF in pseudocode, or a decision box in a flowchart. An algorithm with these features has a logical test. It chooses between two different actions. You need to:
- state the logical test
- state the actions to take if the test result is True (and other actions if it is False).

Repetition
If the algorithm processes a series of numbers (or other data), you may need to use a loop. Remember:
- If you know how many items of data there are, use a counter-controlled loop.
- If you do not know how many items of data there are, use a condition-controlled loop.
- Decide which lines of your algorithm go inside the loop.
- Make sure the loop has an exit condition.

Test yourself
An algorithm was needed to find the smallest of 1000 values. Answer the following questions.
1. Which algorithm that you have worked on already might help you with this task?
2. What type of loop would you use in this algorithm?
3. How many variables do you need? Suggest names for them.
4. What logical test will you include in the algorithm?
5. Using the answers to these questions, create an algorithm for the stated purpose.

Learning activity
Create an algorithm that will count the number of teenagers entering a youth club. The person at the door will record "M" for male, "F" for female and "X" when the club closes. The output will be the number of male and female teenagers who visited the youth club.
Creating software applications
An algorithm produces a solution to a given problem. For example, it may loop until the correct password is entered, or it may produce the average from a group of numbers.

A real-life software system will include many different sub-systems. For example, the software system for a small business may include sub-systems to:

- keep track of the items in stock
- record profit and loss
- pay staff wages.

To create a software system, a programmer must produce lots of smaller sub-systems. Each sub-system does a different task. The sub-systems combine to make software that does all the work needed by the client.

Top-down programming
In top-down programming we begin by analysing the software system into sub-systems. Then we break those sub-systems down into even smaller components. For example, the software to pay staff wages may be broken down further into systems to:

- record staff hours
- calculate pay by multiplying hours by pay per hour
- work out tax and other deductions
- print pay slips.

The programmer will work on these components one at a time. When they are all finished they will be fitted together to make the complete payroll system.

Choice of method
Most programmers nowadays work in teams. The task of making the software is split between different people. Each person creates a different sub-system, then the sub-systems are fitted together.

Advantages
Top-down programming has many advantages:

- It lets you plan the task in full before you begin programming.
- It breaks a big difficult task down into smaller and easier tasks.
- The different tasks can be given to different people, so the work is done more quickly.
- Each sub-system can be tested and perfected before all the sub-systems are fitted together.

Disadvantages
Some programmers do not like using top-down programming. They identify some disadvantages:

- Using top-down design means that coding cannot be the first task – programmers have to start with planning.
- Sometimes the different sub-systems do not work well together and the program goes wrong.

Top-down analysis
You have learned that every computer system:

- starts with input data
- ends with output information.

If you decide to use top-down programming, you must begin by breaking this overall system into sub-systems. For each sub-system you must decide what are its data inputs and information outputs.

Typically, the output from one sub-system will become the input for the next sub-system. A top-down plan must state:

- what all the different sub-systems are
- the inputs and outputs of each sub-system.

A structure diagram shows the different sub-systems and how they relate to each other. Learn more about structure diagrams on pages 248–249.

Learning activity
A programmer wanted to develop a software application to help students keep track of their homework. Explain why.

1. Top-down programming helps with team work. Explain why.
2. Why is testing easier when you use top-down programming?
3. A programmer planned a system by defining sub-systems. What information is included in the plan?
4. A programmer thinks she might be able to break a sub-system down into even smaller sub-systems. Discuss the advantages and disadvantages of this approach.

Test yourself
1. Top-down programming helps with team work. Explain why.
2. Why is testing easier when you use top-down programming?
3. A programmer planned a system by defining sub-systems. What information is included in the plan?
4. A programmer thinks she might be able to break a sub-system down into even smaller sub-systems. Discuss the advantages and disadvantages of this approach.
Solution development

Structure diagrams

Introduction
You have learned that computer systems are made of sub-systems. Structure diagrams show these sub-systems. In this section you will learn to understand structure diagrams and to draw them.

Structure diagram
A computer system is made of smaller sub-systems. When you plan the computer system you decide what the sub-systems are. This is top-down programming.

A structure diagram is a drawing of the computer system. It shows the sub-systems. The first stage of top-down programming is planning the sub-systems. The structure diagram lets you show the plan in visual form.

The structure diagram will be used to:
- plan the work that needs to be done
- tell the program team about the plan
- check the work when it is completed.

Basic structure
A structure diagram looks like a family tree. Every system and sub-system is shown as a box. At the top of the tree is a box with the name of the whole system in it. For example, it may be a payroll system.

Think of the main tasks that must be performed by the system. These tasks are shown in the level below. In this example the payroll system has to record the hours people work, calculate their pay, transfer the money to them, and give them a payslip.

To make sure your structure diagram is correct, you must check the following:
- Have you included every task? There must be nothing missing from the system.
- Are the tasks distinct from each other? There must be no overlap between tasks.

Each task will be assigned to a programmer or a team of programmers. Later, the different sub-systems will be fitted back together to make the finished software system.

Further breakdown
Each of the sub-systems can be broken down into even smaller sub-systems. You carry on with this analysis until you have broken the task down into its smallest components. Each component takes input and processes it to make output.

Test yourself
1. Explain why your structure chart must include every task carried out by the software.
2. Explain why the different sub-systems must be distinct, with no overlap between them.
3. How would you know that the structure chart is completed?
4. Name a software package that you could use to make a structure chart.

Learning activity
Here is some information about a software system. Create a simple structure chart for this software system:
- The stock control system must record all deliveries to the warehouse. It must record all the stock that goes out of the warehouse.
- It must calculate how much stock is left in the warehouse. The software will print out an alert if the level of stock gets too low.

Here is more information about the stock control system. Use this information to expand the structure chart:
- When stock arrives the system will record the time of arrival from the computer clock. The stock details will be entered by hand. Each item of stock will be given a barcode.
- When stock goes out of the warehouse the barcode will be scanned. The stock records will be updated automatically.
Sub-routines

You have learned about top-down programming. Software systems are split up into sub-systems. This helps with planning, development and testing. In top-down programming, programs are broken down into sub-programs. A sub-program is also called a routine or a sub-routine (all these terms mean the same thing). A typical routine will carry out a clearly defined task. There are two types of routine:

- Functions are routines that create an output that can be used by other routines.
- Procedures or sub-procedures are routines that carry out tasks but do not supply any output to other routines.

In Python, functions and procedures are created in the same way. In some programming languages different methods are used to create functions and procedures.

Predefined functions

Some functions are so useful that they come as part of the main Python package. These are called predefined functions. You have used two predefined functions:

- `print()`
- `input()`

These functions are very important. You have used these functions in every program you have written.

Data-type conversion is carried out by predefined functions. You have learned two of these:

- `int()`
- `float()`

There are many other predefined functions in the Python programming language: google "Python predefined function" to see a full list.

Re-usable routines

Programmers will also make their own sub-routines as they complete programming projects. They may re-use these sub-routines in later programming projects. This has important advantages:

- It saves time because programmers can use code that is ready made.
- The programmers have used the code before so they have tested it and know it works.

For example, you made a program that checked a password. This could be used as a sub-routine in other projects. Any time you need a password you could re-use the code you have already made.

Code libraries

Since Python was developed, programmers have written many useful sub-routines. Programmers are helpful and support each other: Programmers will share many of the useful sub-routines they have made. This makes the programming language easier to use. It speeds up program development.

The sub-routines are stored in files called modules. The Python code library stores all the modules that are available. Here is part of the Python code library.

To use these stored modules in a program, you type the word "import" and the name of the module you want to use. From then on you can use any of the procedures and functions from that module in your program.

Can you see the module called "random"? You will use this module now.

Example

The module called "random" includes many useful sub-routines that create random numbers. To use these sub-routines enter this line of code into a program:

```python
import random
```

We will use one of the sub-routines in this module. The sub-routine is called `randint`. This subroutine creates a random integer. The following command will create a random integer between 0 and 100:

```python
random.randint(0, 100)
```

Here is the function in use in a program. A random integer is assigned to a variable called `example`:

```python
example = random.randint(0, 100)
```

Here is a simple program that uses the random module to make a random number, and then shows it on the screen.

```python
# make a random number
import random
example = random.randint(0, 100)
print("the random number is ", example)
```

Test yourself

1. What is the difference between a procedure and a function?
2. What predefined functions have you used in your programming?
3. What are the advantages of re-using code you wrote for a different program?
4. What does the command "import random" do?
**Key terms**

- **Code library**: A set of stored sub-routines which can be used in lots of different programs to make work easier.
- **Effective**: A program which produces the effect that was intended.
- **Logical error**: A program error where the program will be translated into machine code, and the machine code will run, but the program does not work effectively. It does not do what it is supposed to do.
- **Run-time error**: A program error that means that the program can be translated into machine code, but the machine code will not run properly: for example, it crashes or gets stuck in an endless loop.
- **Sub-routine**: A section of program code which carries out a single well defined action. A sub-routine is typically given a name which identifies its purpose. Procedures and functions are sub-routines.
- **Syntax error**: A program error which breaks the rules of the computer language in which the program was written. The program cannot be translated into machine code.
- **Top-down design**: A program development method where a large problem is broken down into manageable parts.
- **Trace table**: A table showing the value of a variable at different stages of an algorithm.
- **Validation**: An error check method where the input data is evaluated using validation criteria. If the data does not meet the criteria an error message is displayed.
- **Verification**: An error check method where the same data is entered twice. The two versions are compared and they should match.

**Project work**

Write a program with the following features:

1. The user inputs a series of numbers. The numbers are stored as a list. The program loops until the user types X.
2. The user then sees a menu which gives them the following choices:
   - A: Count
   - B: Total
   - C: Average
3. The user enters one of the three choices. The program then prints out the entire list, plus the count, total (sum) or average (mean) of the list.

If you complete this easily then add validation checks so all inputs to the list must be numbers from 1–100.