Algorithms

An **algorithm** is a sequence of ordered instructions that are followed step-by-step to solve a problem. This does *not* need to be on a computer.

Decomposition is the breaking down of a complex problem into smaller more manageable problems that are easier to solve.

Abstraction allows us to remove unnecessary detail from a problem leaving us with only the relevant parts of a problem thereby making it easier to solve.

Algorithm Efficiency More than one algorithm can be used to solve the same problem. Normally we use the algorithm that solves the problem in the quickest time with the fewest operations or makes use of the least amount of memory.

Dry run testing is carried out using **trace tables**. The purpose of the trace tables is for the programmer to track the value of the variables and outputs at each step of the program and to track how they change throughout the running of the program.

Flowchart Symbols

We can represent algorithms using flowcharts

Start and Stop

Start | Stop

Process – An operation that the algorithm performs

Process

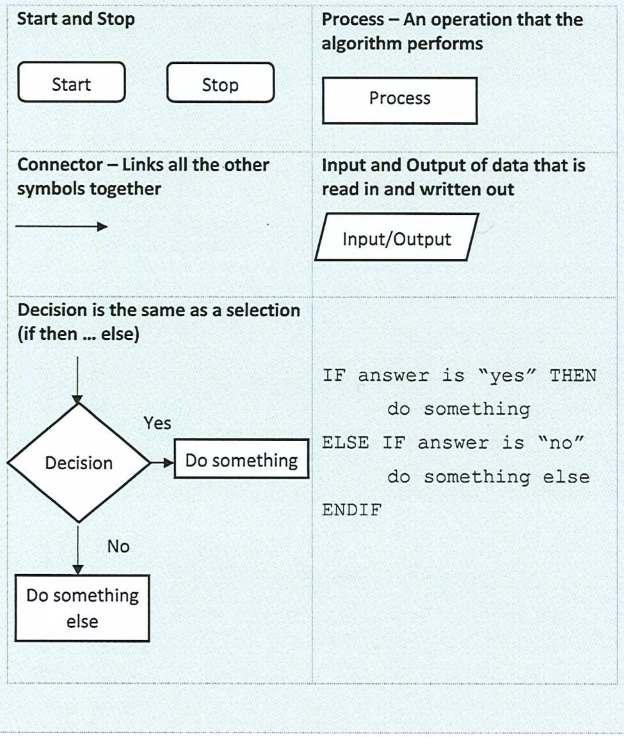
Connector – Links all the other symbols together

→

Input and Output of data that is read in and written out

Input/Output

Decision is the same as a selection (if then ... else)



```
IF answer is "yes" THEN
     do something
ELSE IF answer is "no"
     do something else
ENDIF
```

Pseudocode

We can represent algorithms using pseudocode

	Example	Python equivalent
Variable assignment	a ← 10	`a = 10`
Constant assignment	constant PI ← 3.142	`PI = 3.142`
Input	a ← USERINPUT	`a = input()`
Output	OUTPUT "Bye"	`print("Bye")`
Arithmetic Operators		
Add Multiply Divide Subtract Integer division Modulus (remainder)	+ * / – a ← 7 DIV 2 a ← 7 MOD 2	+ * / – a= 7 // 2 a = 7 % 2
Relational Operators		
Less than Greater than Equal to Not equal to Less than or equal to Greater than or equal to	< > = ≠ or <> ≤ ≥	< > == != <= >=
Boolean Operators		
AND OR NOT	AND OR NOT	AND OR NOT
Selection		

if ..	IF i > 2 THEN
	j ← 10
	ENDIF
		if i > 2:
		j=10

if .. else ...	IF i > 2 THEN
	j ← 10
	ELSE
	j ← 3
	ENDIF
		if i > 2:
		j=10
		else:
		j=3

if ... else if ... else	IF i ==2 THEN
	j ← 10
	ELSE IF i==3 THEN
	j ← 3
	ELSE
	j ← 1
	ENDIF
		if i ==2:
		j=10
		elif i==3:
		j=3
		else:
		j=1

Iteration		
While loops	a ← 1 WHILE a < 4 OUTPUT a a ← a + 1 ENDWHILE	while a<4: print(a) a=a+1
For loops	FOR a ← 0 TO 3 OUTPUT a ENDFOR	for a in range(3): print(a)
Repeat loops	a ← 1 REPEAT OUTPUT a a ← a + 1 UNTIL a←4	
Subroutines		
procedure	SUB hello() OUTPUT "hello" ENDSUB	def hello(): print("hello")

Function (with paramerters and return)	SUB add(n)
	a ← 0
	FOR a ← 0 TO n
	a ← a + n
	ENDFOR
	RETURN a
	ENDSUB
		def add(n):
		a=0
		for a in range(n+1):
		a=a+n
		return a

Built-in functions		
Length of array	LEN(a)	len(a)
Random integer	RANDOM_INT(0, 9)	import random random.randint(0,9)

Sound

Sample - Measure of the analogue signal at a given point in time

Sample rate - number of samples taken per second and is measured in Hertz.

Sample resolution - number of bits used to represent each sample

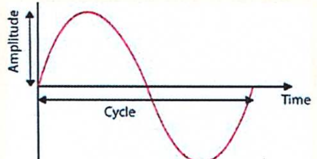
The size of sound files can be calculated using:

size of file = length (seconds) x sample rate x sampling resolution

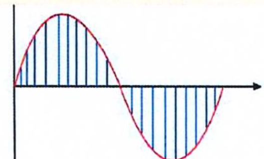
For sound to be stored digitally on a computer it needs to be converted from its continuous analogue form into a discrete binary values. The steps are:

1. Microphone detects the sound wave and converts it into an electrical (analogue) signal
2. The analogue signal is sampled at regular intervals
3. The samples are approximated to the nearest integer (quantised)
4. Each integer is encoded in binary with a fixed number of bits

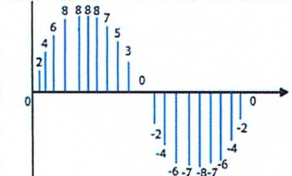
Original analogue signal



Sample signal at regular intervals



Integer values give to each sample



Encode as binary

0 2 4 6 8 8 8 8 7 5 3 0 ->

00000 00010 00100 01000

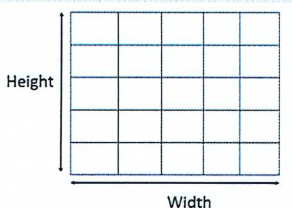
01000 01000 01000 00111

00101 00011 ...

Images

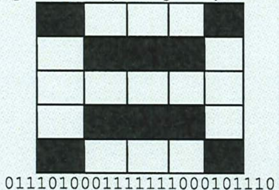
Bitmap images are made up from tiny dots called **pixels**. Each pixel will have a colour associated with it. An image can then be constructed from many of pixels which will have different colours arranged in rows and columns.

Total number of pixels in image = width in pixels x height in pixels



Colour depth is the number of bits used to represent each pixel in an image. If we have a black and white image it has two colours. Each pixel can be represented by a single pixel because a bit value of 0 is black and 1 is white.

Image and corresponding binary encoding



0111010001111111000101110

To represent more colours we can use more bits. For instance if we have 2-bits per pixel we can represent 4 colours because we know have 4 binary code combinations (00, 01, 10 11) where each code represents a different colour

Pixilation occurs when the image is overstretched. In these situations, the image looses quality and has a blocky and blurred appearance. This arises when the image is presented at too large a size and there are not enough pixels to reproduce the details in the image at this larger size.

Calculating the size of a bitmap image

File size in bits = width in pixels x height in pixels x colour depth

File size in bytes = width in pixels x height in pixels x colour depth / 8

Data Compression

The purpose of data compression is to make the files smaller which means that:

- Less time / less bandwidth to transfer data
- Take up less space on the disk

Given that there are 7 bits per ASCII character, the uncompressed size of an ASCII phrase is:

size = number of characters (including spaces) x 7

Run Length Encoding (RLE) is a compression method where sequences of the same values are stored in pairs of the value and the number of those values. For instance, the sequence:

0 0 0 1 1 0 1 1 1 1 0 1 1 1 1

would be represented as:

3 0 2 1 1 0 4 1 1 0 4 1

Huffman coding is a form of compression that allows us to use fewer bits for higher frequency data. More common letters are represented using fewer bits than less common letters. For instance, "a" and "e", which occur in many words would be represented with fewer bit than "z" which occurs rarely.
This allows for much more effective compression than RLE.

The steps involved in Huffman encoding as are follows:

1. Do frequency table
2. Order table
3. Create the tree
4. Add 1, 0 to the branches
5. Encode letters
6. Encode message

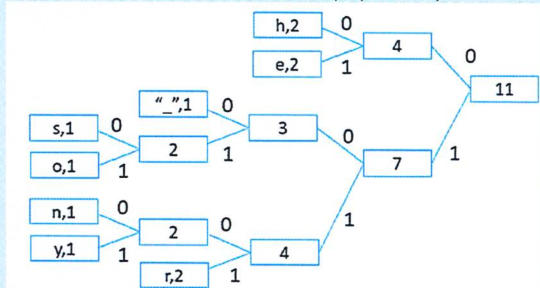
Worked Example: How much smaller is the phrase henry horse encoded using Huffman encoding compared with its uncompressed size.

Calculate the uncompressed size
In the phrase *henry horse* there are 11 characters (including the space). Therefore the uncompressed size is 11 x 7 = 77 bits

Generate ordered frequency table (steps 1 and 2)

letter	frequency
e	2
h	2
r	2
<space>	1
o	1
s	1
y	1
n	1

Create the tree and add 1 and 0 to branches (steps 3 and 4)



Encode letters

Letter	encoding
e	01
h	00
r	111
<space>	100
o	1011
s	1000
n	1100
y	1101

Encode message
00 01 1100 111 1101 100 00 1011 111 1000 01 = 33 bits

Therefore by using compression we have reduced the size from 77 bits to 33 bits a saving of 44 bits.

Data Representation

Number bases

Denary (or decimal) is base-10 and is the number system we are most familiar with. We have the columns of units, tens, hundreds, thousands and so on. Base-10 means that we have 10 possible values (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) in each column.

Binary is base-2 and has 2 values, 0 and 1. It requires a greater number of digits in binary to represent a number than denary. This is how data and instructions are stored in a computer.

To calculate the maximum value for a given number of bits we use 2^n-1 where n is the number of bits. For example for 4 bits we have 2^4-1 which is 15.

Bits	Max value binary	Max value denary
1	1_2	1_{10}
2	11_2	3_{10}
3	111_2	7_{10}
4	1111_2	15_{10}
5	11111_2	31_{10}
6	111111_2	63_{10}
7	1111111_2	127_{10}
8	11111111_2	255_{10}

Hexadecimal is base-16. To make up the 16 values we use the ten denary numbers in addition to 6 letters (A, B, C, D, E, F).

Denary	Hex.	Binary
0_{10}	0_{16}	0000_2
1_{10}	1_{16}	0001_2
2_{10}	2_{16}	0010_2
3_{10}	3_{16}	0011_2
4_{10}	4_{16}	0100_2
5_{10}	5_{16}	0101_2
6_{10}	6_{16}	0110_2
7_{10}	7_{16}	0111_2

Denary	Hex.	Binary
8_{10}	8_{16}	1000_2
9_{10}	9_{16}	1001_2
10_{10}	A_{16}	1010_2
11_{10}	B_{16}	1011_2
12_{10}	C_{16}	1100_2
13_{10}	D_{16}	1101_2
14_{10}	E_{16}	1110_2
15_{10}	F_{16}	1111_2

Hexadecimal is used a lot in computing because it much easier to read than binary. There are far fewer characters than binary. So hexadecimal is often used in place of binary as a shorthand to save space. For instance, the hexadecimal number 7BA3D456 (8 digits) is 01111011101000111101010001010110 (32 digits) in binary which is hard to read.

Hexadecimal is better than denary at representing binary because hexadecimal is based on powers of 2.

Converting between number bases

Denary to binary conversion

1. Create a grid:

128	64	32	16	8	4	2	1

2. Add a 1 to the corresponding cell if number contributes to target number and 0 to all the other cells

Worked example: convert 24_{10} to binary.

128	64	32	16	8	4	2	1
0	0	0	1	1	0	0	0

$16_{10} + 8_{10} = 24_{10}$

The binary value is <u>11000_2</u> (we can ignore the preceding zeros)

Binary to denary conversion

Worked example: Convert 01011001_2 to denary

1. Create the grid:

128	64	32	16	8	4	2	1
0	1	0	1	1	0	0	1

2. Add up the cells that have a corresponding value of 1:
   $64 + 16_{10} + 8_{10} + 1 = \underline{89_{10}}$

Hexadecimal to denary conversion

1) Convert the two hex values separately to denary value
2) Multiply the first value by 16
3) Add the second value

Worked example: Covert $A3_{16}$ to denary

$A_{16} = 10_{10}$
$3_{16} = 3_{10}$
$(10_{10} \times 16_{10}) + 3_{10} = \underline{163_{10}}$

Denary to hexadecimal conversion

1) Integer divide the denary number by 16
2) Take the modulus 16 of the denary number
3) Convert the two numbers to the corresponding hex values.

Worked example: Convert 189_{10} to hex

$189_{10} / 16_{10} = 11_{10}$ remainder 15_{10}
$11_{10} = B_{16}$
$15_{10} = F_{16}$
$189_{10} = \underline{BF_{16}}$

Hexadecimal to binary conversion

1. Find the corresponding 4-bit binary number for the two numbers
2. Concatenate the two binary values to give the final binary value

Example: convert $C3_{16}$ to binary

$C_{16} = 12_{10} = 1100_2$
$3_{16} = 3_{10} = 0011_2$
$\underline{11000011_2}$

Binary to hexadecimal conversion

1. Split the binary number into groups of 4 bits: 1110_2 1010_2
2. Find the corresponding Hex value for each of the 4-bit groups

Worked example: Convert 11101010_2 to hexadecimal

1110_2 | 1010_2
$1110_2 = 14_{10} = E_{16}$
$1010_2 = 10_{10} = A_{16}$
$\underline{EA_{16}}$

Units of Information

Unit	Symbol	Number of bytes
Kilobyte	KB	10^3 (1000)
Megabyte	MB	10^6 (1 million)
Gigabyte	GB	10^9 (1 billion)
Terabyte	TB	10^{12} (1 trillion)

A bit is the fundamental unit of binary numbers. A bit is a binary digit that can be either 0 or 1.

1 byte = 8 bits
1 nibble = 4 bits

Character Encoding

Character coding schemes allows text to be represented in the computer. One such coding scheme is **ASCII**. ASCII uses 7 bits to represent each character which means that a total of 128 characters can be represented.

Lower case letters	26
Upper case letters	26
Numbers	10
Symbols (e.g. comma, colon)	33
Control characters	33

ASCII encoded values for some characters

A	1000001_2	65_{10}
B	1000010_2	66_{10}
a	1100001_2	97_{10}
b	1100010_2	98_{10}
"0"	0110000_2	48_{10}
"1"	0110001_2	49_{10}

- ASCII has a limited character set (7 bits, 128 characters), but **Unicode** has 16 bits and allows many more (65K) characters.
- Unicode provides a unique character for different languages and different platforms.
- It allows us to represent different alphabets for instance Greek, Mandarin, Japanese, Emojis etc.
- Unicode and ASCII are the same up to 127.

Binary addition

Binary addition rules

$0_2 + 0_2 = 0_2$
$0_2 + 1_2 = 1_2$
$1_2 + 0_2 = 1_2$
$1_2 + 1_2 = 10_2$ (carry 1)
$1_2 + 1_2 + 1_2 = 11_2$ (carry 1)

Example

```
    1 0 1 0 1 0 0 1₂
    0 0 0 0 1 0 0 1₂
+   0 0 0 1 0 1 0 1₂
    1 1 0 0 0 1 1 1₂
carry 1 1 1     1
```

Binary Shift

The binary shift operator is used to perform multiplication and division of numbers by powers of 2

multiply/divide	x 16	x 8	x 4	x 2	/ 2	/ 4	/ 8
shift	<<4	<<3	<<2	<<1	>>1	>>2	>>3

Example: Apply shift operator to 1101_2 (13_{10})

Shift	Result	denary
<<1	11010_2	$13_{10} \times 2_{10} = 26_{10}$
<<2	110100_2	$13_{10} \times 4_{10} = 52_{10}$
>>1	110	$13_{10} // 2_{10} = 6_{10}$

Note that odd numbers are rounded down to the nearest integer when the right shift operator is applied.

Programming - Python

Comment – Text within the code that is ignored by the computer. A Python comment is preceeded by a #.

```
# This is an example of a comment
```

Output – Processed information that is sent out from a computer

Python	Pseudocode
print("Hello World!") Hello World! print("Hello", "World!") Hello World! print("Hello"+"World!") HelloWorld! print("Hello\nWorld!") Hello World!	OUTPUT "Hello World"

Input – Data sent to a computer to be processed

print("Enter name")	OUTPUT "Enter name"
name=input()	name ← USERINPUT
print("Hello", name)	OUTPUT "Hello", name
print("Enter age")	OUTPUT "Enter age"
age=int(input())	age ← USERINPUT

Assignment - The allocation of data values to variables, constants, arrays and other data structures so that the values can be stored.

- *Variable* – Value that can change during the running of a program. By convention we use lower case to identify variables (eg a=12)
- *Constant* – Value that remains unchanged for the duration of the program. By convention we use upper case letters to identify constants. (e.g. PI=3.141)

Data Types

Integer	age = 12	age ← 12
Float (real) number	height = 1.52	height ← 12
Character	a = 'a'	a ← 'a'
String – multiple characters	name = "Bart"	name ← "Bart"
Boolean (true/false)	a = True b = False	a ← True b ← False

Arithmetic Operators

Add	7 + 2	= 9	7 + 2
Subtract	7 - 2	= 5	7 - 2
Multiply	7 * 2	= 14	7 * 2
Divide	4 / 2	= 2	4 / 2
power	2 ** 3	= 8	2 ** 3
Integer division	7 // 2	= 3	7 DIV 2
Modulus (remainder)	7 % 2	= 1	7 MOD 2

Relational Operators – Allows the Comparison of values

Less than	<	<	7<2	-> False
Greater than	>	<	7 > 2	-> True
Equal to	==	==	7==2	-> False
Not equal to	!=	≠ or <>	7!=2	-> True
Less than or equal to	<=	≤	7<=2	-> False
Greater than or equal to	>=	≥	7>=2	-> True

Boolean Operators

AND	and	7 < 2 and 1 < 2	-> False
OR	or	7 < 2 or 1 < 2	-> False
NOT	not	not 7 < 2	-> True

Sequencing represents a set of steps. Each line of code will have some operation and these operations will be carried out in order line-by-line

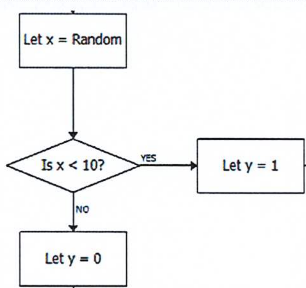
Using + operator for adding a = 1 b = 2 c = a + b print(c) -> 3	 a ← 1 b ← 2 c ← a + b OUTPUT c
Using + operator for concatenation a = 'Hello ' b = 'World' c = a + b print(c) -> Hello World	 a ← 'Hello ' b ← 'World' c ← a + b OUTPUT c

Random number

Random integer	import random random.randint(0,9)	RANDOM_INT(0,9)
Choice	random.choice('a','b','c')	
Random value from 0 to 1	random.random()	

Selection represents a decision in the code according to some condition. The condition is met then the block of code is executed otherwise it is not. Often alternative blocks of code are executed according to some condition.

```
x=RANDOM_INT()
IF  x < 10 THEN
  y=1
ELSE
  y=0
ENDIF
```



IF ...	IF i > 2 THEN
	  j ← 10
	ENDIF
		if i > 2:
		  j=10

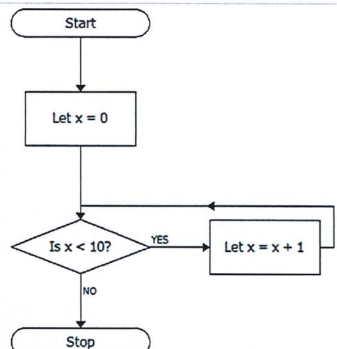
IF ... ELSE ...	IF i > 2 THEN
	  j ← 10
	ELSE
	  j ← 3
	ENDIF
		if i > 2:
		  j=10
		else:
		  j=3

IF ... ELSE IF ... ELSE	IF i ==2 THEN
	  j ← 10
	ELSE IF i==3
	  j ← 3
	ELSE
	  j ← 1
	ENDIF
		if i ==2:
		  j=10
		elif i==3:
		  j=3
		else:
		  j=1

Iteration Sometimes we wish the code to repeat a set of instructions

WHILE loops are used when the we do not know beforehand the number of iterations needed and this varies according to some condition.

```
x = 0
while (x < 10):
    x = x + 1
```



while True:
  print("Hello World")
	WHILE TRUE
	  OUTPUT "Hello World"
	ENDWHILE

a=0
while a<4:
  print(a)
  a=a+3
	a ← 0
	WHILE a < 4
	  OUTPUT a
	  a ← a + 3
	ENDWHILE

FOR loops are used when we know before hand the number of iterations we wish to make.

for a in range(3):
  print(a)
	FOR a ← 0 TO 3
	  OUTPUT a
	ENDFOR

Nested structures - Use constructs (e.g. `WHILE, FOR, IF`) inside another.

use a nested FOR loop to print out a grid	`for i in range (10):`
	` for i in range (10):`
	`  print ("x ",end="")`
	` print()`

Use a nested while and if to print out only even numbers	`i=0`
	`while i<51:`
	` if (i%2==0):`
	`  print(i)`
	` i=i+1`

Lists

Create a list	`shapes=["square","circle"]`
Access element by index pos	`shapes[1] -> circle`
Append item to list	`shapes.append("triangle")`
Remove item from list	`shapes.remove("circle")`
Remove item from list by index	`shapes.pop(1)`
Insert item into list	`shapes.insert(2,"rectangle")`
Number of elements in a list	`len(shapes)`
Get index pos of item in list	`shapes.index("triangle")`

Concatenating lists	`shapesGroup1["square","circle"]`
	`shapesGroup2=["triangle"]`
	`shapes=shapesGroup1+shapesGroup2`

Loop through list	`for i in range(len(shapes)):` ` print(shapes[i])`
Reverse elements in a list	`shapes.reverse()`
Order elements in a list	`shapes.sort()`

2D lists - A list if lists

Create a 2D list	`d = [ [23, 14, 17], [12, 18, 37], [16, 67, 83]]`

Another way to create a 2D list	`a = [23, 14, 17]`
	`b = [12, 18, 37]`
	`c = [16, 67, 83]`
	`d = [a,b,c]`

Access element by index position	`d[1][2] -> 37`

Strings

Get length of a string	`len("Hello")`	`LEN("Hello")`
Character to character code	`ord("a") -> 97`	`ORD("a")`
Character code to character	`chr(101) -> 'e'`	`CHR(101)`
String to integer	`a=int("12")`	`a=INT("12")`
String to float	`a=float("12.3")`	`a=FLOAT("12.3")`
integer to string	`a=str(12)`	`a=STR(12)`
real to string	`a=str(12.3)`	`a=STR(12.3)`

Concatenation -merge multiple strings together	`a="hello "`
	`b="world"`
	`c=a+b`
	`print(c) ->`
	`hello world`

Return the position of a character If there is more than 1 of the same character the position of the first character is returned.	`student = "Hermione"` `student.index('i')`
Find the character at a specified position	`student = "Hermione"` `print(student[2]) -> r`

sub strings - select parts of a string

Example	`student="Harry Potter"`	
Output the first two characters	`print(student[0:2])`	Ha
Output the first three characters	`print(student[:3])`	Har
Output characters 2-4	`print(student[2:5])`	Rry
Output the last 3 characters	`print(student[-3:])`	Ter
Output a middle set of characters	`print(student[4:-3])`	y Pot

*A negative value is taken from the end of the string.

Subroutines are a way of managing and organising programs in a structured way. This allows us to break up programs into smaller chunks.

- Can make the code more modular and more easy to read as each function performs a specific task.
- Functions can be reused within the code without having to write the code multiple times.

- **Procedures** are subroutines that do not return values
- **Functions** are subroutines that have both input and output

Procedure: No input parameters or return	`SUB greeting()`
	` OUTPUT "hello"`
	`ENDSUB`
		`def greeting():`
		` print("hello")`
		
		`call: greeting()`

Procedure: One input parameter, no return	`SUB greeting(name)`
	` OUTPUT "Hello",name`
	`ENDSUB`
		`def greeting(name):`
		` print("Hello",name)`
		
		`greeting("grey")`

Function: 1 input parameter, and 1 return value	`SUB add(n)`
	` a ← 0`
	` FOR a ← 0 TO n`
	`  a ← a + n`
	` ENDFOR`
	` RETURN a`
	`ENDSUB`
		`def add(n):`
		` a=0`
		` for a in range(n+1):`
		` a=a+n`
		` return a`

Function: Two input parameters, and 1 return value	`SUB (num1,num2)`
	` sum=num1+num2`
	` return sum`
		`def add(num1,num2):`
		` sum=num1+num2`
		` return sum`
		
		`greeting(1,2)`

The **scope** of a variable determines which parts of a program can access and use that variable.

A **global variable** is a variable that can be used anywhere in a program. The issue with global variables is that one part of the code may inadvertently modify the value because global variables are hard to track.

A **local variable** is a variable that can only be accessed within a certain block of code typically within a function. Local variables are not recognized outside a function unless they are returned. There is no way of modifying or changing the behavior of a local variable outside its scope.

Global variables need to defined throughout the running of the whole program. This is an inefficient use of memory resources. Local variables are defined only when they are needed an so have less demand on memory. Local variables only exist within the subroutine.

Reading and writing files

Open file Whatever we are doing to a file whether we are reading, writing or adding to or modifying a file we first need to open it using:

```
open(filename,access_mode)
```

There are a range of access mode depending on what we want to do to the file, the principal ones are given below:

Access Mode	Description
r	Opens a file for reading only
w	Opens a file for writing only. Create a new file if one does not exist. Overwrites file if it already exists.
a	Append to the end of a file. Create a new file if one does not exist.

Reading text files

read – Reads in the whole file into a single string	`f=open("filetxt","r")`
	`print(f.read())`
	`f.close()`

readline – Reads in each line one at a time	`f=open("file.txt","r")`
	`print(f.readline())`
	`print(f.readline())`
	`print(f.readline())`
	`f.close()`

readlines – Reads in the whole file into a list	`f=open("file.txt","r")`
	`print(f.readlines())`
	`f.close()`

Writing text files

Write in single lines at a time	`file=open("days.txt",'w')`
	`file.write("Monday\n")`
	`file.write("Tuesday\n")`
	`file.write("Wednesday\n")`
	`file.close()`

Write in a list	`say=["How\n","are\n","you\n"]`
	`file=open("say.txt",'w')`
	`file.writelines(say)`
	`file.close()`

Data Validation Routines

Check if an entered string has a minimum length	`OUTPUT "Enter String"`
	`s ← USERINPUT`
	`IF LEN(S) > 5 THEN`
	`  OUTPUT "STRING OK"`
	`ELSE`
	`  OUTPUT "TOO SHORT"`
	`ENDIF`

Check is a string is empty	`OUTPUT "Enter String"`
	`s ← USERINPUT`
	`IF LEN(S) == 0 THEN`
	`  OUTPUT "EMPTY STRING"`
	`ENDIF`

Check if data entered lies within a given range	`OUTPUT "Enter number" s num ←`
	`USERINPUT`
	`IF num > 1 AND num < 10`
	`  OUTPUT "Within range"`
	`ENDIF`

Authentication Routine

```
OUTPUT "Enter Username"
username ← USERINPUT
OUTPUT "Enter Password"
password ← USERINPUT

WHILE username != "bart" OR password !="abc"

 OUTPUT "Login failed"
 OUTPUT "Enter Username"
 username ← USERINPUT
 OUTPUT "Enter Password"
 password ← USERINPUT

ENDWHILE

OUTPUT "Login Successful"
```

Debugging

Syntax errors – Errors in the code that mean the program will not even run at all. Normally this is things like missing brackets, spelling mistakes and other typos.

Runtime errors – Errors during the running of the program. This might be because the program is writing to a memory location that does not exist for instance. eg. An array index value that does not exist.

Logical errors - The program runs to termination, but the output is not what is expected. Often these are arithmetic errors.

Test data

Code needs to be tested with a range of different input data to ensure that it works as expected under all situations. Data entered need to be checked to ensure that the input values are:

- within a certain range
- in correct format
- the correct length
- The correct data type (eg float, integer, string)

The program is tested using normal, erroneous or boundary data.

Normal data - Data that we would normally expect to be entered. For example for the age of secondary school pupils we would expect integer values ranging from 11 to 19.

Erroneous data - Data that are input that are clearly wrong. For instance, if some entered 40 for the age of a school pupil. The program should identify this as invalid data but at the same time should be able to handle this sensibly which returns a sensible message and the program does not crash.

Boundary data - Data that are on the edge of what we might expect. For instance if someone entered their age as 10, 11, 19 or 20.