**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.

**Procedures** are subroutines that do not return values

**Functions** are subroutines that have both input and output and **return values**

**Variable** is an identifier / name of a memory location used to store data that can change during runtime

**Constant** is an identifier / name of a memory location used to store data, that that remains unchanged for the duration of the program.

```
GLOBAL globalMultiplier = 2  // A global variable used for modifying results

PROCEDURE calculateArea(length, width)
    area = length * width  // Local variable for area calculation
    print( "Inside CalculateArea:")
    print( "Length: ", length)
    print( "Width: ", width)
    print( "Area: ", area)
END PROCEDURE

FUNCTION doubleArea(originalArea)
    doubled = originalArea * global Multiplier  // Uses global variable
    print( "Doubled Area: ", doubled)
    return doubled
END FUNCTION

// Main program
print( "Inside Main Program:")

length = float(input())
width = float(input())

calculateArea(length, width)
print(caluclatedArea)
doubledResult = doubleArea(areaResult)
print(doubledResult)
```

**Parameter** – An item of data that is passed to a subroutine when it is called and is used as a variable within the subroutine.

**Arguments** – The values assigned to parameters when a subroutine is called.

**Passing by Value**
A copy of the value is passed to the subroutine and discarded at the end
Its value outside of the subroutine remains unaffected
**Passing by Reference**
Address of parameter is given to the subroutine
Value of the parameter will be updated at the given address

**Score of variables**
**Scope is the section of code in which the variable can be accessed**
A local variable within a subroutine takes can only be accessed within the subroutine, whereas global variables can be accessed across the whole program.

**Local Variables:**

- Only work inside the subroutine where they are created.
- Deleted when the subroutine finishes.
- Keep subroutines independent and organized.

**Global Variables:**

- Can be used anywhere in the program.
- Helpful for shared values.
- Risk of accidental changes.
- Stay in memory until the program ends, using more space.

**Data Types –** determines what value a variable can hold and the operation that can be performed on a variable

| | | |
|---|---|---|
| *Integer* | age = 12 | A whole number |
| *Float (real)* | height = 1.52 | A number with a decimal point |
| *Character* | a = 'a' | A single letter, number or symbol |
| *String* | name = "Bart" | Multiple characters |
| *Boolean* | a = True<br>b = False | Has two values; true or false |

# Programming Constructs

## Sequence:

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

## 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.

**If statement**

| **Pseudocode** | **Python** |
|---|---|
| entry = input("Enter day of the week: ")<br>if entry==”a” then<br>print(“You selected A”)<br>elseif entry==”b” then<br>print(“You selected B”)<br>else<br>print(“Unrecognised selection”)<br>endif | entry = input("Enter day of the week: ")<br>if entry==”a”:<br>print(“You selected A”)<br>elif entry==”b”:<br>print(“You selected B”)<br>else:<br>print(“Unrecognised selection”) |

**Swich Case Statement**

```
entry = input("Enter day of the week: ")
switch entry:
    case “A”:
        print(“You selected A”)
    case “B”:
        print(“You selected B”)
    default:
        print(“Unrecognised selection”)
endswitch
```

**Not supported in python**

# Iteration

**Iteration** Do a set of statements multiple times. Iteration is either '**count controlled**' or '**condition controlled**'.

- Count repeats the section n times,
- Condition waits until a condition has been met before stopping iteration

## Count Controlled

**Execute a sequence of statements multiple times**

| Pseudocode | Python |
|---|---|
| for i=0 to 7<br>  print("Hello")<br>next i<br><br>Will print hello 8 times (0-7inclusive). | for i in range(8):<br>  print("Hello") |

## While – (Condition Controlled)

Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

| Pseudocode |
|---|
| while answer!="computer"<br>  answer=input("What is the password<br>endwhile |

| Python |
|---|
| while answer!="computer":<br>  answer=input("What is the password |

## do until Loop (Condition Controlled)

Like a while statement, except that it tests the condition at the end of the loop body.

| Pseudocode | Python |
|---|---|
| do<br>  answer=input("What is the password?")<br>until answer=="computer" | Not supported in python |

## 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 |

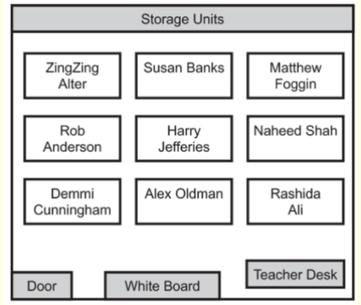
## Boolean Operators

| | | | |
|---|---|---|---|
| AND | and | 7 < 2 and 1 < 2 | -> Fa |
| OR | or | 7 < 2 or 1 < 2 | -> Fa |
| NOT | not | not 7 < 2 | -> Tru |

## 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 |

**Exam style question**
Sally is a classroom teacher. She would like a program to be able to organise where students will sit in her classroom.
A plan of her classroom is shown **here.**




Sally would like to increase the security of her program by adding a password to enter the program. She has created the procedure, checkPassword, to do this.

```
01   procedure checkPassword()
02       correctPassword = "ComputerScience12"
03       check = false
04       while check == false
05          enteredPassword = input("Enter Password")
06          if enteredPassword == correctPassword then
07             check = true
08          endif
09       endwhile
10   endprocedure
```

1. Identify the programming construct used on lines 06 to 08 in the procedure checkPassword.

   Selection

   **[1]**

2. Sally has used a while loop on line 04 of the procedure checkPassword.

   Explain why Sally has used a while loop instead of a for loop. [2]

   - The number of user attempts is not known
   - The code will need to continue until the password entered is correct
   - A while loop will keep repeating until the correct password has been input // condition is met
   - A for loop will only repeat a certain number of times
   - A for loop may continually ask for password even though it's been entered correctly

   **[3]**

3. Sally could have used a do until loop instead of a while loop.

   Rewrite lines 04 to 09 of the procedure checkPassword using a do until loop instead of a while loop.

```
do
   enteredPassword=input("Enter Password")
   if enteredPassword == correctPassword then
      check = true
   endif
until check == true
```

- Correct use of do at the start of the loop.
- Correct use of until at the end of the loop.
- Correct logic for inputting password, checking the entered password and for setting check to true/checking the password within the condition of the loop.

A function, toBinary(), is needed to calculate the binary value of a denary integer between 0 and 255.

toBinary() needs to:
- take an integer value as a parameter
- divide the number by 2 repeatedly, storing a 1 if it has a remainder and a 0 if it doesn't
- combine the remainder values (first to last running right to left) to create the binary number
- return the binary number.

For example, to convert 25 to a binary number the steps are as follows:

| | |
|---|---|
| 25 / 2 = 12 | remainder 1 |
| 12 / 2 = 6 | remainder 0 |
| 6 / 2 = 3 | remainder 0 |
| 3 / 2 = 1 | remainder 1 |
| 1 / 2 = 0 | remainder 1 |

return value = 11001
Write the function toBinary().

You should write your function using pseudocode or program code. [6]

**Answer:** 1 mark per bullet to max 6
- function header taking parameter
- looping appropriately e.g. until value is 0
- dividing by 2 and finding remainder e.g. MOD
- adding 1 or 0 correctly
- …appending to a value to be returned // final string reversed
- reducing value to use within loop
- returning calculated value

**Pseudocode**

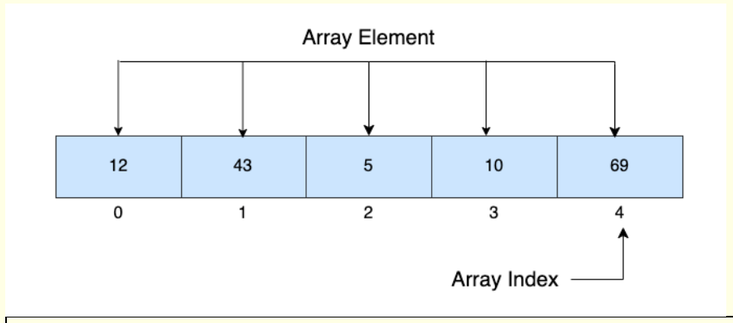
```
function toBinary(denary)
    binaryValue=""
    while denary > 0
        temp = denary MOD 2
        binaryValue = str(temp) + binaryValue
        denary = denary DIV 2
    endwhile
    return binaryValue
endfunction
```

**Python**

```
def toBinary(denary):
    binaryValue=""
    while denary > 0:
        temp = denary % 2
        binaryValue = str(temp) + binaryValue
        denary = denary // 2

    return binaryValue
```

## Data Structures




Allows multiple items of data to be stored under one identifier
Can store a table structure
Reduces need for multiple variables

## Array

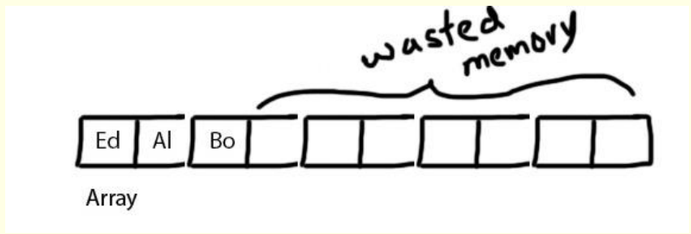
**Fixed size** (static)

**Same data type** for all elements

**Fast access** using index positions

**Mutable** (elements can be changed, added, removed)

**Not as flexible** as lists (cannot grow/shrink)




Example:

birdName = ["robin", "blackbird", "pigeon", "magpie"]

birdName[2] = "pigeon" #the index here is 2

numSpecies = len(birdName) #will assign 4 to numSpecies.

**2-dimensional arrays**
An array can have two or more dimensions.

Imagine a 2-dimensional array called numbers, with 3 rows and 4 columns.

Elements in the array can be referred to by their row and column number, so that:

**numbers[1,3] = 8 in the example below.**

| | Column 0 | Column 1 | Column 2 | Column 3 |
|---|---|---|---|---|
| Row 0 | 1 | 2 | 3 | 4 |
| Row 1 | 5 | 6 | 7 | 8 |
| Row 2 | 9 | 10 | 11 | 12 |

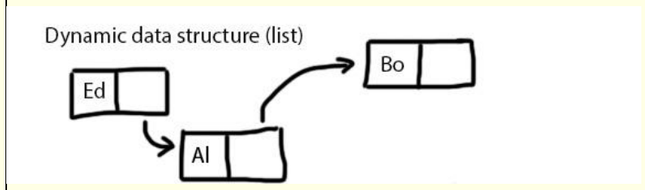
## List

**Dynamic size** (can grow and shrink)

**Can store different data types** in the same list

**Mutable** (elements can be changed, added, removed)

**Slightly slower** than arrays in some operations due to dynamic memory allocation




## Tuple

**Fixed size**

Immutable (values cannot change_

**Can store different data types**

**More memory-efficient & faster** than lists for iteration

**Cannot be modified** (no adding, removing, or changing elements)

colour = (255, 0, 0)  # Red color

## Exam style question

A 2-dimensional (2D) array, data, holds numeric data that Karl has entered. The declaration for the array is:

array data[16,11]

The array data, has 16 'rows' and 11 'columns'.

This is an extract from data:

| | 0 | 1 | 2 | 3 | … | 10 |
|---|---|---|---|---|---|---|
| 0 | 1 | 5 | 7 | 12 | … | 36 |
| 1 | 3 | 4 | 15 | 16 | … | 48 |
| 2 | 0 | 0 | 1 | 3 | … | 10 |
| 3 | 12 | 16 | 18 | 23 | … | 100 |
| … | … | … | … | … | … | … |
| 15 | 6 | 10 | 15 | 25 | … | 96 |

The data in each 'row' is in ascending numerical order.

Karl needs to analyse the data.

Karl needs to find the mean average of each 'column' of the array. The mean is calculated by adding together the numbers in the column, and dividing by the quantity of numbers in the column.

| | | | |
|---|---|---|---|
| 1 | 5 | 7 | 12 |
| 3 | 4 | 15 | 16 |
| 0 | 0 | 1 | 3 |
| 12 | 16 | 18 | 23 |

For example, the first 'column' mean would be: (1+3+0+12)/4 = 4

Write an algorithm to output the mean value of each 'column' in the array data. [5]

1 mark per bullet

- Looping through each column [1]
- Looping through each row [1]
- …Adding to a total [1]
- …Calculating average correctly [1]
- Outputting average [1]

| OCR Pseudocode | Using in range |
|---|---|
| for col =0 to 10<br>  total = 0<br>  for row = 0 to 15<br>  total = total + data[row,col]<br>  next row<br>  print("Average", total/16)<br>  next y | for col in range (11):<br><br>  total = 0<br>  for row in range(16):<br>  total = total + data[row,col]<br>  print("Average", total/16) |

A card game uses a set of 52 standard playing cards. There are four suits; hearts, diamonds, clubs and spades. Each suit has a card with a number from; 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13.

The card game randomly gives 2 players 7 cards each. The unallocated cards become known as the deck.

The players then take it in turns to turn over a card. A valid move is a card of the same suit or the same number as the last card played.

The winner is the first player to play all of their cards.
The cards are held in the 2D array cards. The first index stores the card number and the second index stores the suit, both as strings.

Write a **pseudocode statement** or program code to declare the array cards.

array cards[7, 4] or array towns [3,8]

## Records

| | | |
|---|---|---|
| A record is a data structure that groups together related items of data<br>You can store more than one type of data together | **Array of Records** | **Record** \| **Class** |

A record is an unordered data structure
Can have multiple instances

Player records:

Player1 record

| Olivia |
|---|
| 35 |

Player 2 record

| Luke |
|---|
| 40 |

**Creating a record structure**

recordStructure recordstructurename

*fieldname : datatype*
…
endRecordStructure

The pseudocode to define a new complex data structure called player:

RECORD player
name: String
score: int
ENDRECORD

*Adding data to the record*

*recordidentifier : recordstructurename*
*recordidentifier.fieldname = data*

The pseudocode to define a new complex data structure called Player:

Player1: player
Player1.player.name = 'Olivia'
Player1.player.score = 35

**Records** are treated as **data types**, so they can be held within a single **array**.

This allows for storage of more than one **record** within the same structure. This structure is essentially an **array of records**.

The table below simplifies the way that this data would be held in memory.

- The records for the players could be stored in a 1D array.
- It allows easy access/indexing/manipulation of each data item in turn
- 1D Array can hold multiple items of same data type – record
- Maximum number of array elements is known

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| Olivia<br>35 | Luke<br>40 | Adam<br>25 | Alex<br>35 |

**Pseudocode of Arrays of records:**

Players(100) As player

We can then reference any element of the array:

Players(3).name = "Jane"

| Similarities | | |
|---|---|---|
| Data structure | A record is a data structure that stores data together, organised by attributes. | A *class* is a record with associated methods. Each object is a data structure with attributes stored together |
| Set up in advance | Attributes and structure for the record are set up. Meaning that it is created by the programmer for a particular purpose. | Constructor method defines the class object |
| Store data of different types | `recordStructure` **`pets`**<br>`name` **`: String`**<br>**`type : String`**<br>**`age : Int`**<br>**`endRecordStructure`** |  |
| Both can have multiple instances | Yes | Yes |
| Accessed by their names | Yes | Yes |
| Differences | | |
| Class also has methods<br>Class can include visibility of properties / private | | |

## Working with strings

| Strings | | Exam style Question | Mark scheme |
|---|---|---|---|
| Get length of a string | len("Hello") | The function validate Answer takes in the randomLetters as an array of letters and the player's answer as a string. It then | - Function traverses every letter of answer (1)<br>– Function traverses every randomLetters (1) |

| | |
|---|---|
| Character to character code | asc("a") |
| Character code to character | CHR(101) |
| String to integer | a=INT("12") |
| String to float | a=FLOAT("12.3") |
| integer to string | a=STR(12) |
| real to string | a=STR(12.3) |

**substrings** - 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.

checks if the word the player has entered only contains letters from the 10 random letters with each letter being used only once. (At this stage the program doesn't check if the answer provided is an actual word.) It then returns a score, out of 10, for a valid word or 0 for an invalid word.

Example:
If the random letters are: OPXCMURETN
The word COMPUTER returns 8
Whereas
The word POST returns 0 (there is no S in the random letters).
The word RETURN returns 0 (there is only one R in the random letters).

**Complete the function validateAnswer**

**function validateAnswer(answer, randomLetters[]) [6]**

– Correctly checks each letter of answer against each of randomLetters (1)
– Returns 0 if answer contains a letter that doesn't occur in randomLetters (1)
– Returns 0 if letter occurs more times in answer than randomLetters (1)
– Returns answer length for a valid word.(1)

**Solution:**

```
function validateAnswer(answer, randomLetters)
    valid = True
    score = 0
    lengthAnswer = len(answer)
    lengthRandom = len(randomLetters)
    index = 0
    while (valid == True) and (index < lengthAnswer)
        numChar = 0  # Initialise numChar for each letter
        currentLetter = answer[index:1]
        for i = 1 to lengthRandom-1
            if currentLetter == randomLetters[i]
                numChar = numChar + 1
        # If the letter does not exist in randomLetters list
        if numChar == 0
            valid = False
        index = index + 1  # Move to the next letter in answer
    # If all letters pass check, answer is valid
    if valid == True
        score = lengthAnswer
    return score
print(validateAnswer("POST", "OPXCMURETN")
```

## Working with 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)

**Exam style Question**

Albert runs a competition each week in his local village hall. So that Albert can contact the winner he would like a program for the winner to enter their telephone number which is then checked and written into a text file.

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")<br>print(f.read())<br>f.close() |
| readline – Reads in each line one at a time | f=open("file.txt","r")<br>print(f.readline())<br>print(f.readline())<br>print(f.readline())<br>f.close() |
| readlines – Reads in the whole file into a list | f=open("file.txt","r")<br>print(f.readlines())<br>f.close() |

**Writing text files**

| | |
|---|---|
| *Write in single lines at a time* | file=open("days.txt",'w')<br>file.write("Monday\n")<br>file.write("Tuesday\n")<br>file.write("Wednesday\n")<br>file.close() |
| Write in a list | say=["How\n","are\n","you\n"]<br>file=open("say.txt",'w')<br>file.writelines(say)<br>file.close() |

The rules for Albert's program are as follows:

1. The telephone number is entered. This is checked to ensure that the first digit is a 0
2. If the first digit is not a 0 then a message saying "Needs To Start With 0" is printed
3. If the first digit is a 0 then the telephone number is passed into a pre-existing function called checkLength as a parameter. This will return true if the length of the telephone number is long enough
4. If the telephone number is long enough then it is written into a text file called "winner.txt"
5. If the telephone number is not long enough then a message saying "Not Long Enough" is printed.

Complete the procedure competitionWinner so that it meets the rules of Albert's program.

You should write your procedure using pseudocode or program code.

```
procedure competitionWinner() [5]

    telNum = input("Enter Telephone Number")
    if telNum[0] == "0" then
          length = checkLength(telNum)
       if length == true then
                myfile = openWrite("winner.txt")
                myfile.write(telNum, 'w')
                myfile.close()
        else
             print ("Not Long Enough")
        endif
     else
        print ("Needs To Start With 0")
     endif

Endprocedure
```

**Answer:** 1 mark per bullet up to a maximum of 5 marks, e.g.:

- Suitable logic for inputting the telephone number
- Suitable logic for ensuring the telephone number starts with a 0
- Suitable logic for passing the telephone number into the function checkLength
- If true, suitable logic for opening and closing winner.txt
- …suitable logic for writing the telephone number to winner.txt
- Suitable logic for printing "Needs To Start With 0" and "Not Long Enough"

## Recursive algorithms

**Recursion**
When a subroutine calls itself during execution.
Continues until a stopping condition is met. (Base Case)

Recursion produces the same result as iteration, but is more suited to certain problems which are more easily expressed using recursion.

**Example**

A common example of a naturally recursive function is **factorial**, shown below:

- The Factorial of a positive integer, n, is defined as the product of the sequence n, n-1, n-2,
- The factorial of a number is the product of all the integers from 1 to that number.
- For example, the factorial of 3 (denoted as 6!) is 1*2*3= 6
- Factorial is not defined for negative numbers and the factorial of zero is one, 0! = 1

**Iterative Solution**

```
Function factorial(number):
    factorial = 1
    # loop from the number till 1
    for number in range(number, 1,-1):
        # multiply current number with the
product of previous numbers
        factorial = factorial * number
    return fact
```

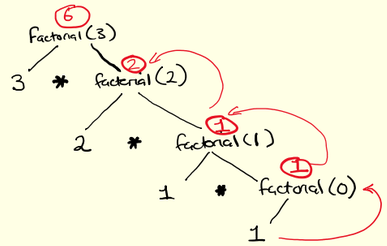
**Recursive Solution**

```
function factorial(number)
    if number == 0 or 1 then #base case
        return 1
    else:
        return number * factorial(number - 1);
    endif
end function
```

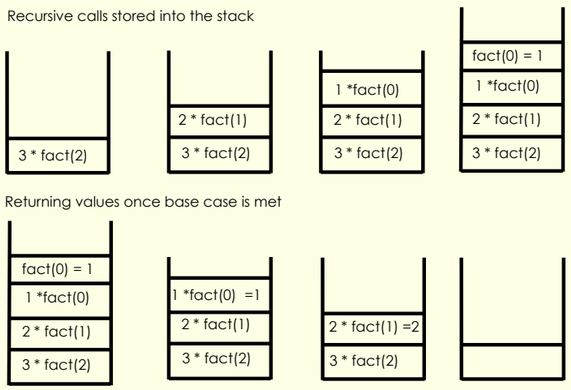
*Example: Trace code where n=3*

| *Call* | *n* | *Return* | |
|---|---|---|---|
| *factorial(3)* | *3* | *6* | *3*(Factorial (2))* |
| *factorial(2)* | *2* | *2* | *2 * (factorial (1)* |
| *factorial(1)* | *1* | *1* | *1 *(factorial(0))* |
| *factorial (0)* | *0* | | |

*Final answer* ***6***




Each time the function calls itself, a new stack frame is created within the call stack , where parameters , local variables and return addresses are stored.




**Benefit**

- **Easier to read and understand** – The logic often follows the problem's natural structure.
- **Concise and quick to write** – Requires fewer lines of code than iterative solutions.
- **Ideal for specific problems** – Particularly useful for tree structures and divide-and-conquer algorithms.
- **Breaks problems into smaller parts** – Each recursive call simplifies the problem step by step.

**Drawbacks of Recursion:**

- **Risk of stack overflow** – Too many recursive calls can exceed memory limits, causing a crash.
- **Requires a well-defined base case** – Without a proper stopping condition, recursion can lead to infinite loops.
- **Harder to trace and debug** – Each call has its own set of variables, making execution more complex to follow.
- **Higher memory consumption** – Recursive calls store additional data on the stack, using more memory than iteration.
- **Often slower than iteration** – Maintaining the call stack adds extra processing overhead.

**Exam Style Questions**

- Re-write an iterative algorithm into a recursive algorithm (vice versa)
- Discuss the benefits and drawbacks of recursive algorithms compared to iterative algorithms.

**High Level Programming Languages**

• Much easier to learn, write and debug.
• Examples include Python, Java and C
• Code written in these languages must be translated to machine code before it can be executed.

**Advantages**
• Much more widely understood and used.
• Easier to learn, code in and understand.
• Much quicker to produce usable code.
• More support and learning resources are available.
• Easier to debug and find issues

**Disadvantages**
• Less flexible.
• Must be translated before being executed
• Very difficult to write and understand.
• Much more time consuming to produce code.

**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.

**Programming Standards**

- No function may be longer than a single page of code: this is to reduce complexity and aid readability.
- Variable identifiers must conform to a standard convention: this helps others to understand the code and reduces the likelihood of duplication, makes maintenance easier.
- Each function must have a single entry point: this reduces complexity and makes the search for any bugs more straightforward.
- Variables must not be set up outside the scope of a function: this sets a limit on where to look for bugs and reduces the likelihood of a problem spread across many modules.
- Indentation
- Global variables – use upper case UPPER_CASE_WITH_UNDERSCORES
- In python - function names should be lowercase
- Comment your code - Do not do line-by-line comments - makes the code look almost unreadable

**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.

**IDE and Debugging**

**Source code editor**
The editor aims to make the coding process easier by providing features such as autocompletion of words, indentation, syntax highlighting

Programming Techniques KO

A programmer uses an Integrated Development Environment (IDE).

Identifying **and** describing **three** IDE features that can help the programmer to develop or debug a program.

**[6]**

**Syntax highlighting**… to identify keywords, variables and help identify syntax errors

**Breakpoints**…stop a program running at a point to check variables

**Variable watch window**…view how variables change while the program executes

example.py

```
1  def add_numbers(a, b):
2      """This function adds two numbers and returns the result."""
3      result = a + b
4      print("The result is ", result)
5
6  def multiply_numbers(a, b):
7      """This function multiplies two numbers and returns the result."""
8      result = a * b
9      print("The result is ", result)
10
11 # Main program
12 num1 = float(input("Enter first number: "))
13 num2 = input("Enter second number: ")
14
15 # Call the subroutines
16 sum_result = add_numbers(num1, num2)
17 product_result = multiply_numbers(num1, num2)
```

Shell

```
%Run example.py
Enter first number: 23
Enter second number: 43
Traceback (most recent call last):
  File "/Users/elizabethallgar/Library/CloudStorage/OneDrive-KingJames'sSchool/example.py", line 16, in <module>
    sum_result = add_numbers(num1, num2)
  File "/Users/elizabethallgar/Library/CloudStorage/OneDrive-KingJames'sSchool/example.py", line 3, in add_numbers
    result = a + b
TypeError: unsupported operand type(s) for +: 'float' and 'str'
>>>
```

Variables

| Name | Value |
|---|---|
| add_numbers | <function add_numbers at 0x1012e6cb0> |
| multiply_numbers | <function multiply_numbers at 0x10133ce50> |
| num1 | 23.0 |
| num2 | '43' |

**Error diagnostics**… to locate and fix errors

**Auto-complete**… start typing a command/identifier and it completes it

```
rand_number = random.
```

- choice
- getrandbits
- randint
- random
- randrange
- seed
- uniform

**Stepping / step through**… run the program line by line to check variable values at each stage

example.py

```
1  def add_numbers(a, b):
2      """This function adds two numbers and retu
3      result = a + b
4      print("The result is ", result)
5
6  def multiply_numbers(a, b):
7      """This function multiplies two numbers an
8      result = a * b
9      print("The result is ", result)
10
11 # Main program
12 num1 = float(input("Enter first number: "))
13 num2 = float(input("Enter second number: "))
14
15 # Call the subroutines
16 sum_result = add_numbers(23.0, 12.0)
17 product_result = multiply_numbers(num1, num2)
```