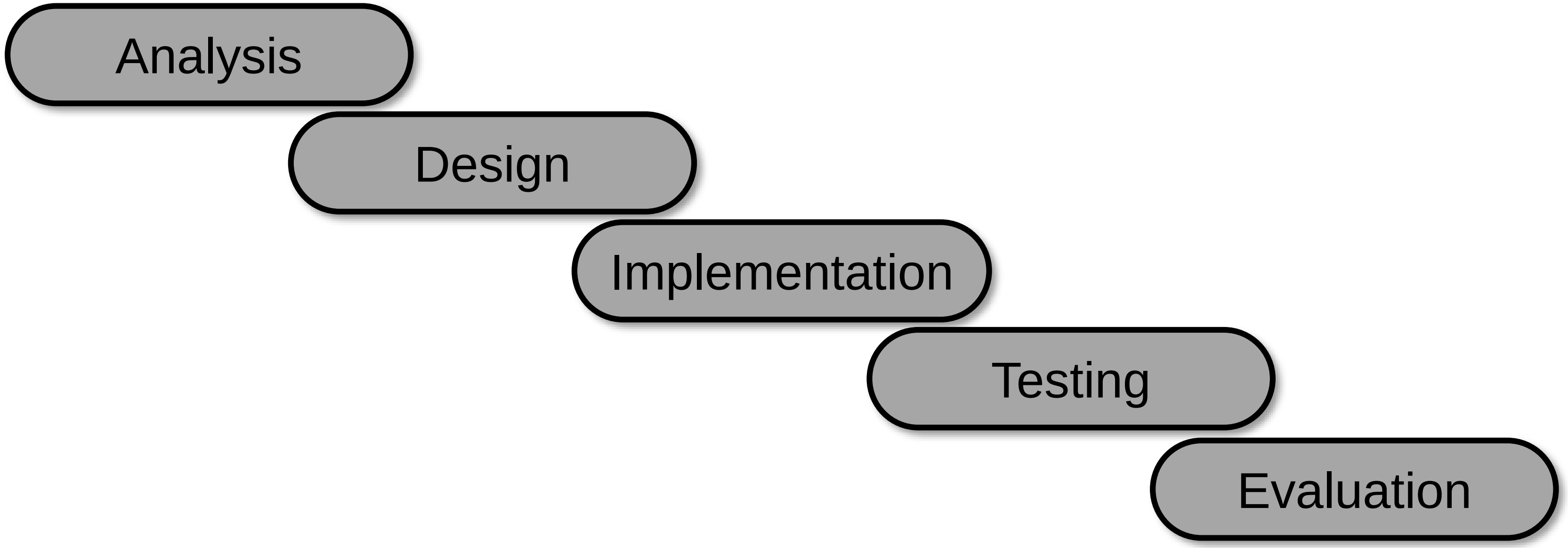
# Software Design and Development

# Development Methodologies

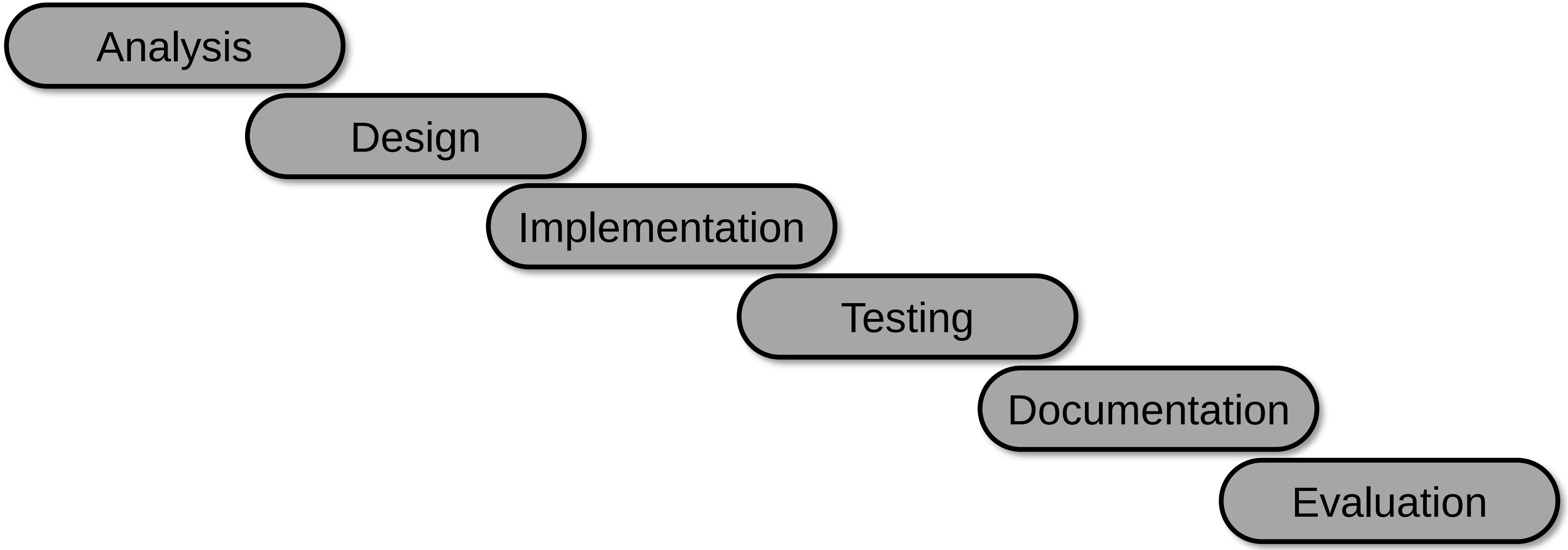
# Development Process

As we’ve learned in previous units, when developing, software teams will work through different phases of development.


Analysis
Design
Implementation
Testing
Evaluation

# Development Process

There is one more step that we have to be aware of during Software development: Documentation.



Analysis

Design

Implementation

Testing

Documentation

Evaluation

# Roles Within a Development Team

Development teams are usually made up of different people with different roles, and each of these roles  is usually responsible for one of the development phases.

### Client

the person/company the software is being built for

### Analyst

the link between the development team and the client

### Designer

Responsible for designing the software system

### Developer

carries outs the implementation of software

### Tester

checks the completed program against the design

# Analysis Phase

During the **Analysis** phase, the Analyst will gather requirements from the Client. It can be difficult for clients to communicate what they need systems to be able to do, so the Analyst will use different techniques to gather requirements to make sure these are as accurate as possible.

In the Analysis phase the team should produce the program purpose and the functional requirements (inputs, processes and outputs).

# Design Phase

During the **Design** phase the Designer will take the user requirements and functional requirements and turn this into a program plan.

In the Design phase the team should produce a list of **variables** and **data types**, a **user interface design** and a program plan, showing the algorithms that make up the program (this could be a **flowchart**, a **structure diagram** or **pseudocode**).

# Implementation Phase

During the **Implementation** phase, the Developer will turn the program plan into code.

Developers will also test their code to make sure that it runs. Developers are usually responsible for resolving **syntax errors** and **runtime errors** as part of the Implementation phase.

# Testing Phase

During the **Testing** phase the Tester will run the program multiple times to check that it works, comparing the code to the program plan created during the **Design** phase.

Testers will use normal, extreme and exceptional test data to find **logic errors**.

In the Testing phase, the team should produce a **testing table** with all the tests that have been run and whether they passed.

# Documentation Phase

During the **Documentation** phase, documentation is created to help IT teams to install the software (technical guides) or to help users to use the software (user guides or tutorials).

In the Documentation phase, the **team** should produce **technical guides**, **user guides** and **tutorials**.

**Internal commentary** is another type of documentation, but that is completed by the **Developer** during the **Implementation** phase.

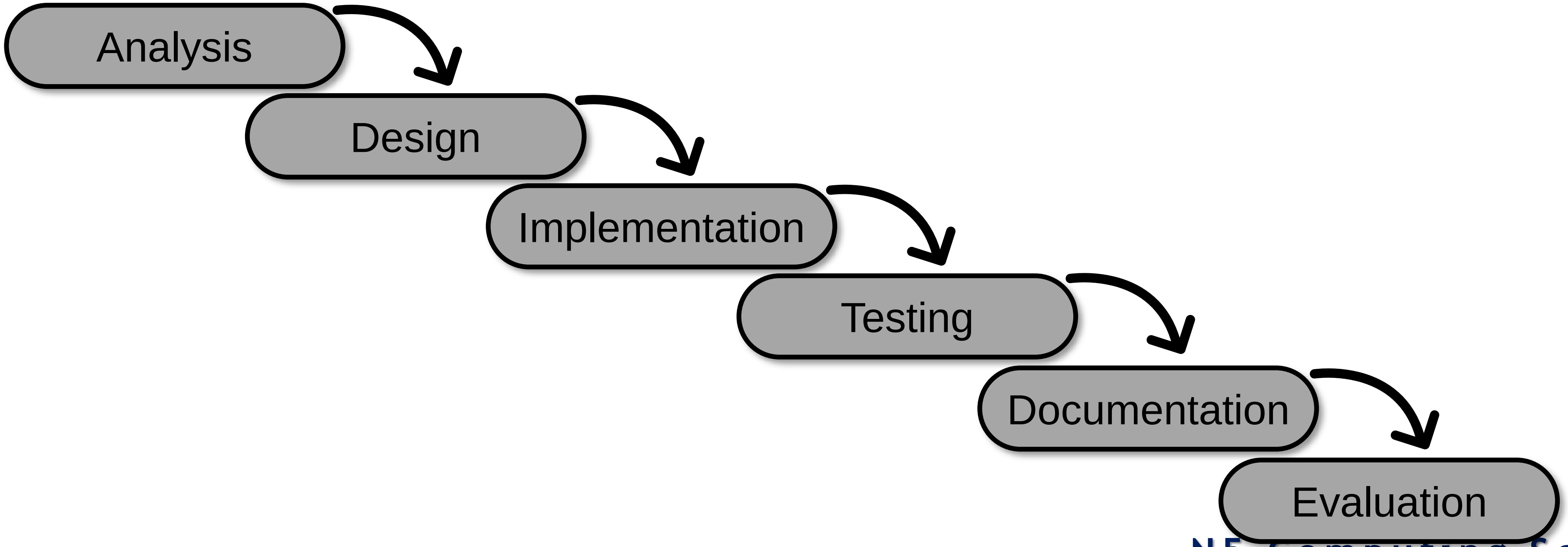
# Evaluation Phase

During the **Evaluation** phase, several areas are checked.

- **Fitness for purpose**: the client checks if the program meets the requirements identified during the Analysis phase
- **Efficiency**: efficient programs use appropriate constructs that reduce the demand on RAM.
- **Robustness**: robust programs can cope with unexpected values without crashing.
- **Readability**: Describes how easy it is to read and edit code, particularly by a different developer. Readable code uses meaningful variable names, white space and internal commentary.

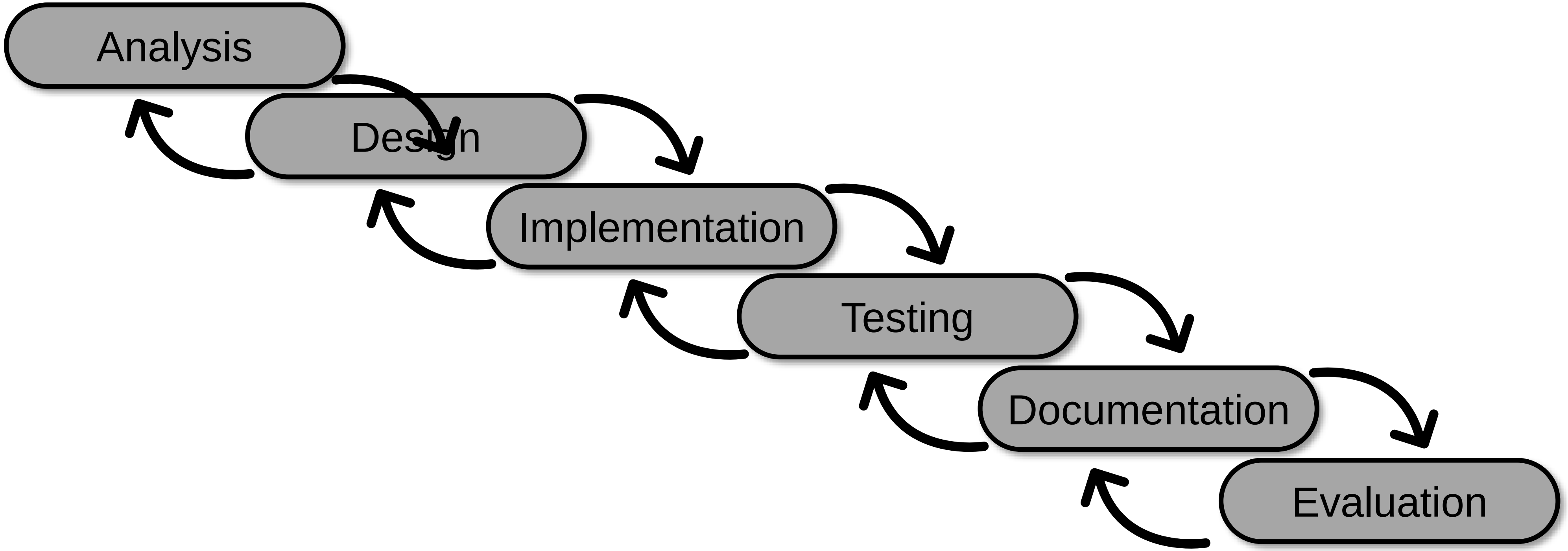
# Development Process

These steps are carried out in a deliberate order, called a **process**. The most traditional process is called the **waterfall process**, as it resembles a waterfall when visualised:



# Iterative Development Process

This development process is referred to as iterative as teams can repeat steps if a problem is found. For example, if the testers find a bug, the team can return to and repeat the Implementation phase.


Analysis

Design

Implementation

Testing

Documentation

Evaluation

# Iterative Development Process

Iteration, or repeating phases, can be very time consuming and expensive. If the client identifies a problem in the Evaluation then the team may have to go right back to the Analysis phase.

It’s important that each phase is completed carefully during the first iteration so that the teams can minimise how many times the teams have to iterate.

# Analysis

# Analysis Overview

During the Analysis phase of Software Design and Development we need to consider two areas:

- the purpose of the software
- the functional requirements of the software

# Program Purpose

The purpose of a piece of software is often expressed as a description of what the software will be used for, which can be used to help create functional requirements.

# Functional Requirements

Functional requirements are defined in terms of inputs, process and outputs:

- What information will be input?
- What process should be applied to these variables? (often a calculation, decision, or use of a pre-defined functions)
- What information will be displayed to the user?

# Analysis Example

Program Purpose

A class of 20 pupils is raising money for charity. For each pupil, the program will ask how much money they raised, validate that it is a value larger than 0, and then add the amount to the total. When there are no pupils left, the program should decide if the class has won a prize by raising more than £150, displaying a congratulations message if they have. The program will then display the total amount raised.

# Analysis Example

## Functional Requirements

| Inputs | Processes | Outputs |
|---|---|---|
| Amount raised by pupil | Validate amounts raised | Congratulations message |
| | Calculate total | Total |
| | Decide if class have won a prize | |

# Design

# Design

During the Design phase of Software Design and Development we need to:

- identify the data types and structures required for a problem
- design a program using one of various different design techniques

# Variables

Variables are used to store a single piece of data.

When creating variables we store 2 pieces of information: the name and the value

```
score = 215
```

name value

# Variable Data Types

There are 5 data types we need to know for National 5:

| Data Type | Type of information stored |
|---|---|
| Character | A single letter or symbol |
| String | Multiple letters or symbols |
| Numeric (Integer) | A positive or negative whole number |
| Numeric (Real) | A positive or negative decimal number |
| Boolean | True or False |

# Arrays

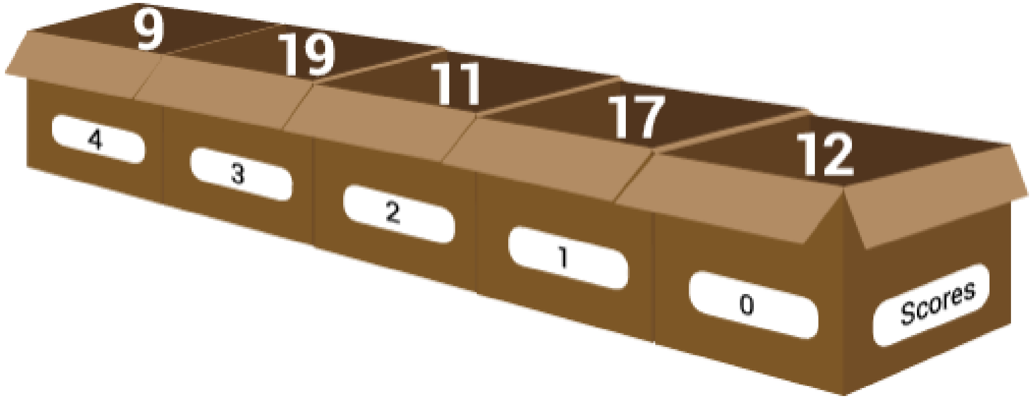
Arrays are a list of values of the same data type.

Arrays are used to stored related pieces of information e.g. we could store names in one array, and test scores in a different array.

# How do arrays work?

An array has one name but multiple values.

We give each value a number, called an **index**, which represents the value’s position within the array.



Arrays begins at index 0, so the first item added to the array is stored in index 0.

# User Interface Design

A user interface design shows how the user is going to interact with your code, i.e. what the user of your program will see when they run your program.

In Python we have been providing the user with text prompts, so our user interfaces will include any prompts we write to get information from the user, or any information that we will display to the user.

# User Interface Design

Program Purpose

A class of 20 pupils is raising money for charity. For each pupil, the program will ask how much money they raised, validate that it is a value larger than 0, and then add the amount to the total. When there are no pupils left, the program should decide if the class has won a prize by raising more than £150, displaying a congratulations message if they have. The program will then display the total amount raised.

# User Interface Design

Prompt (computer) Response (user)

How much money did you raise? ________

How much money did you raise? ________

How much money did you raise? ________

How much money did you raise? ________

How much money did you raise? ________

How much money did you raise? ________

Congratulations, your class raised more than £150!

Your class raised a total of £______

# Program Planning

Program planning is creating a solution to the program purpose. During this stage you create algorithms (i.e. deciding what steps are needed to solve the problem)

There are 3 design notations we need to be able to read and understand in National 5:

- Pseudocode
- Structure Diagrams
- Flowcharts

# Pseudocode

Pseudocode is a written design notation of the steps needed to solve a problem. It is not based on a programming language, meaning you don’t have to worry about syntax.

Pseudocode should:

- define the main steps of a program
- Refine/break down the main steps where possible (not all main steps need refined)
- have indentation to help identify loops and selection statements

# Pseudocode Example

## Program Purpose

A class of 20 pupils is raising money for charity. For each pupil, the program will ask how much money they raised, validate that it is a value larger than 0, and then add the amount to the total. When there are no pupils left, the program should decide if the class has won a prize by raising more than £150, displaying a congratulations message if they have. The program will then display the total amount raised.

# Pseudocode Example

```
SET total TO 0
FOR LOOP 1 TO 20
    Get valid amount from user
    SET total TO total + amount
END LOOP
Decide if class gets a prize
Display total
```

# Pseudocode Example

```
1.  SET total TO 0
2.  FOR LOOP 1 TO 20
3.      Get valid amount from user
4.      SET total TO total + amount
5.  END LOOP
6.  Decide if class gets a prize
7.  Display total
```

Refinements

```
3.1 RECEIVE amount FROM USER
3.2 WHILE amount < 0
3.3     SEND error message TO DISPLAY
3.4     RECEIVE amount FROM USER
3.5 END LOOP

6.1 IF total > 150 THEN
6.2     SEND “Congratulations” TO DISPLAY
6.3 END IF
```

# Pre-Defined Functions in Pseudocode

```
SET score TO ROUND (score, 2)
SET numberOfCharacters TO LENGTH (firstname)
SET bonusBall TO RANDOM(1, 59)
```

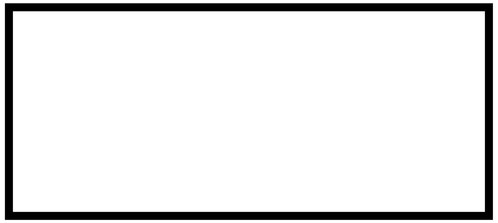
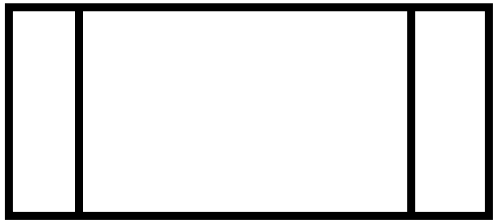
# Structure Diagrams

Structure diagrams are a visual representation of the steps needed to solve a problem.

Structure diagrams are read from the top down, from left to right.

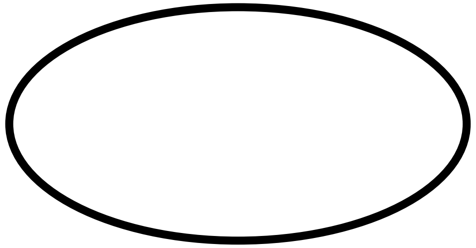
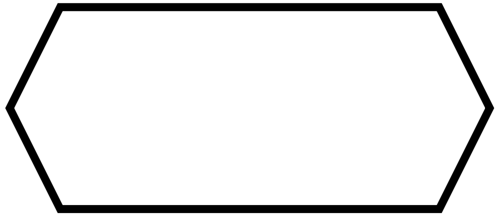
# Structure Diagram Symbols

The following symbols are used in structure diagrams:

| Symbol | Name | Use |
|---|---|---|
|  | Process | Used to show that a process is needed (such as user input, a calculation or displaying information) |
|  | Pre-Defined Function | Used to show that a pre-defined function will be used (such as random, round or length) instead of a developer writing the process themselves. |

# Structure Diagram Symbols

The following symbols are used in structure diagrams:

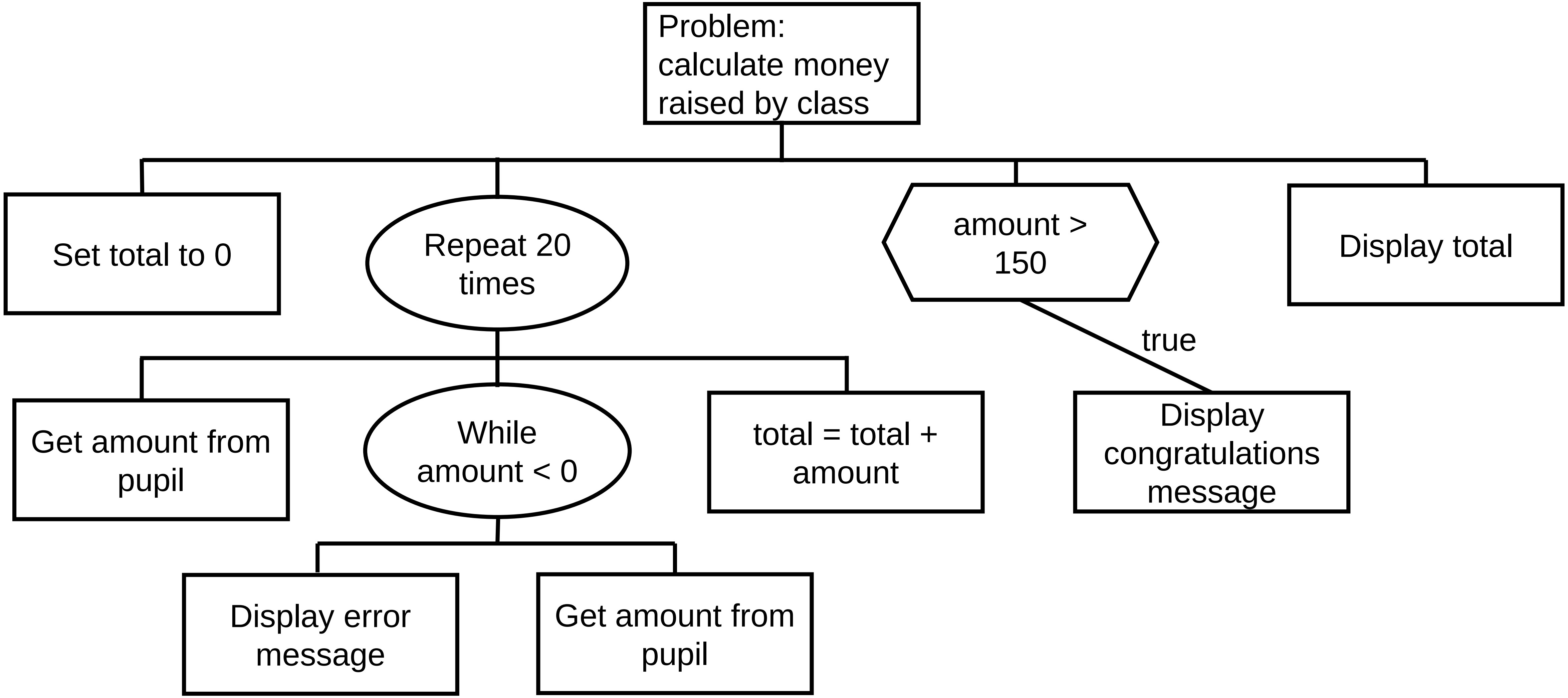
| Symbol | Name | Use |
|---|---|---|
|  | Loop | Used to show that code should be repeated. This is used for both fixed and conditional loops |
|  | Selection | Used to show that the program needs to decide which path to follow i.e. different code should be executed for different scenarios |

# Structure Diagram Example

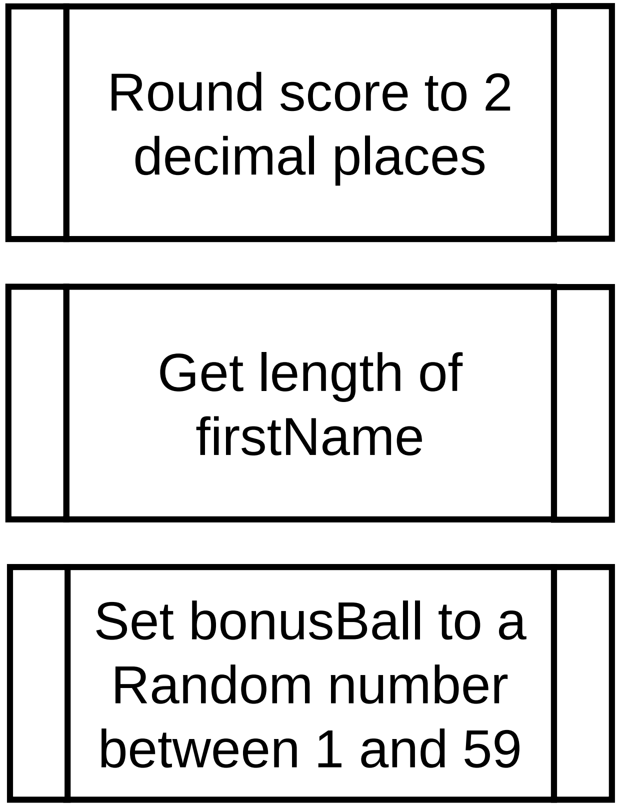
Program Purpose:

A class of 20 pupils is raising money for charity. For each pupil, the program will ask how much money they raised, validate that it is a value larger than 0, and then add the amount to the total. When there are no pupils left, the program should decide if the class has won a prize by raising more than £150, displaying a congratulations message if they have. The program will then display the total amount raised.

# Structure Diagram Example



# Pre-Defined Functions in Structure Diagrams



Round score to 2 decimal places

Get length of firstName

Set bonusBall to a Random number between 1 and 59

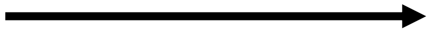
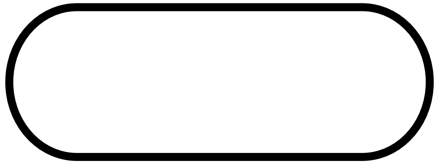
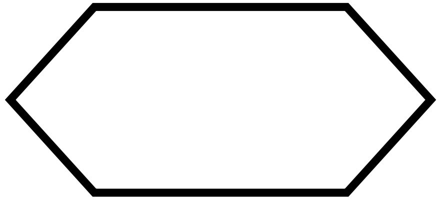
# Flowcharts

Flowcharts are another visual data technique that are used to represent the flow of data through a program.

Flowcharts are read by following the flow line (arrow).

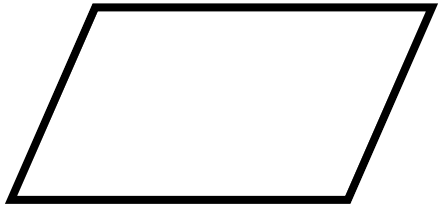
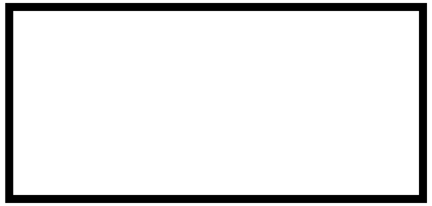
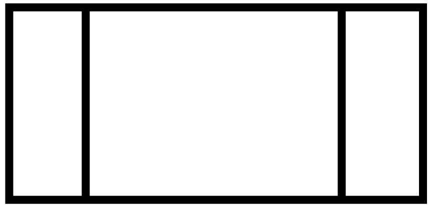
# Flowchart Symbols

The following symbols are used in flowcharts:

| Symbol | Name | Use |
|---|---|---|
|  | Flow line | Shows the direction of flow between symbols |
|  | Terminal | Shows the start and end of the program |
|  | Initialisation | Shows the declaration of a variable, and assignment to an initial value (i.e. set total to 0 for a running total algorithm) |

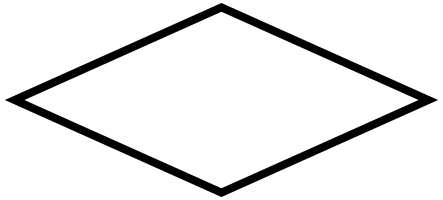
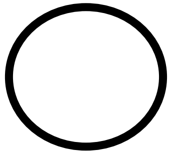
# Flowchart Symbols continued…

The following symbols are used in flowcharts:

| Symbol | Name | Use |
|---|---|---|
|  | Input/Output | Shows data is input or output |
|  | Process | Used to show that a process is needed, e.g. a calculation |
|  | Pre-Defined Function | Used to show that a pre-defined function will be used (such as random, round or length) instead of a developer writing the process themselves. |

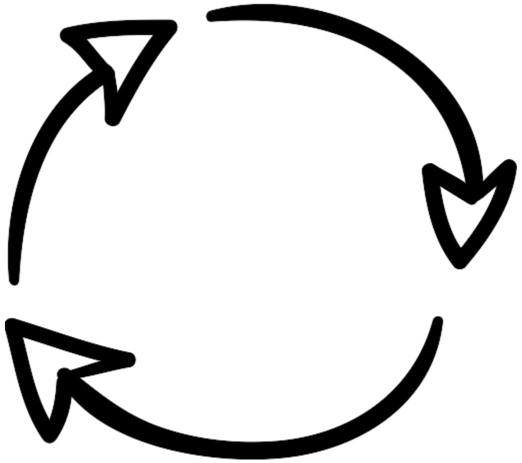
# Flowchart Symbols continued…

The following symbols are used in flowcharts:

| Symbol | Name | Use |
|---|---|---|
|  | Decision | Shows a decision has to be made, with branches for different outcomes. Often used for conditional loops and selection statements |
|  | Connector | Used if you are running out of room on the page so you can keep it all on one page |

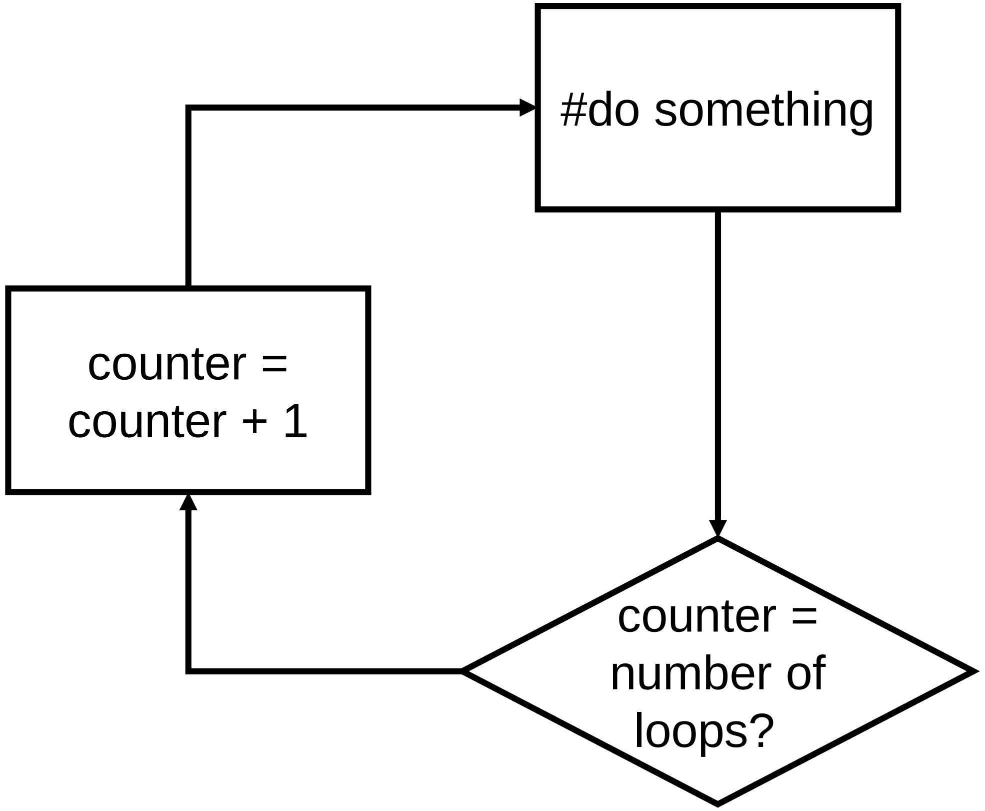
# Flowchart Loops

There is no dedicated symbol for a loop because of the way a flow chart is read – by following the arrow. If the arrows returns to a symbol that has already executed then a loop should be used.

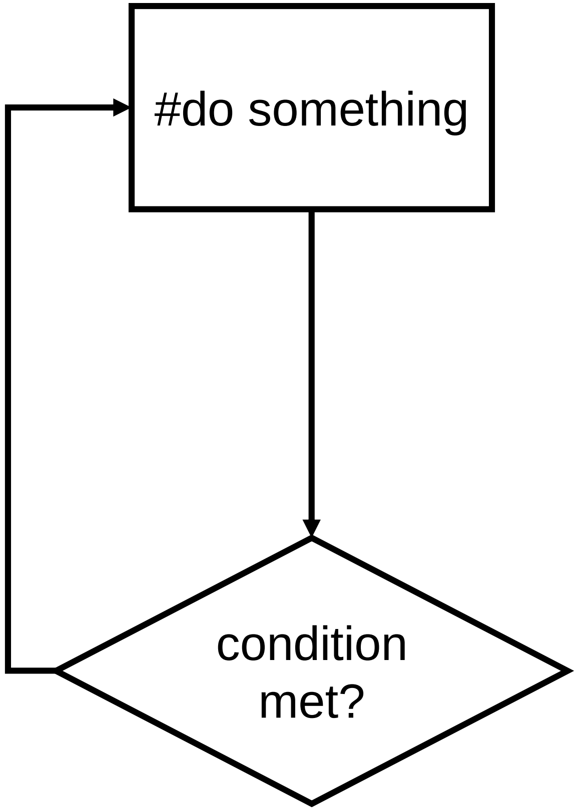


# Flowchart Loops

## Fixed Loop



## Conditional Loop

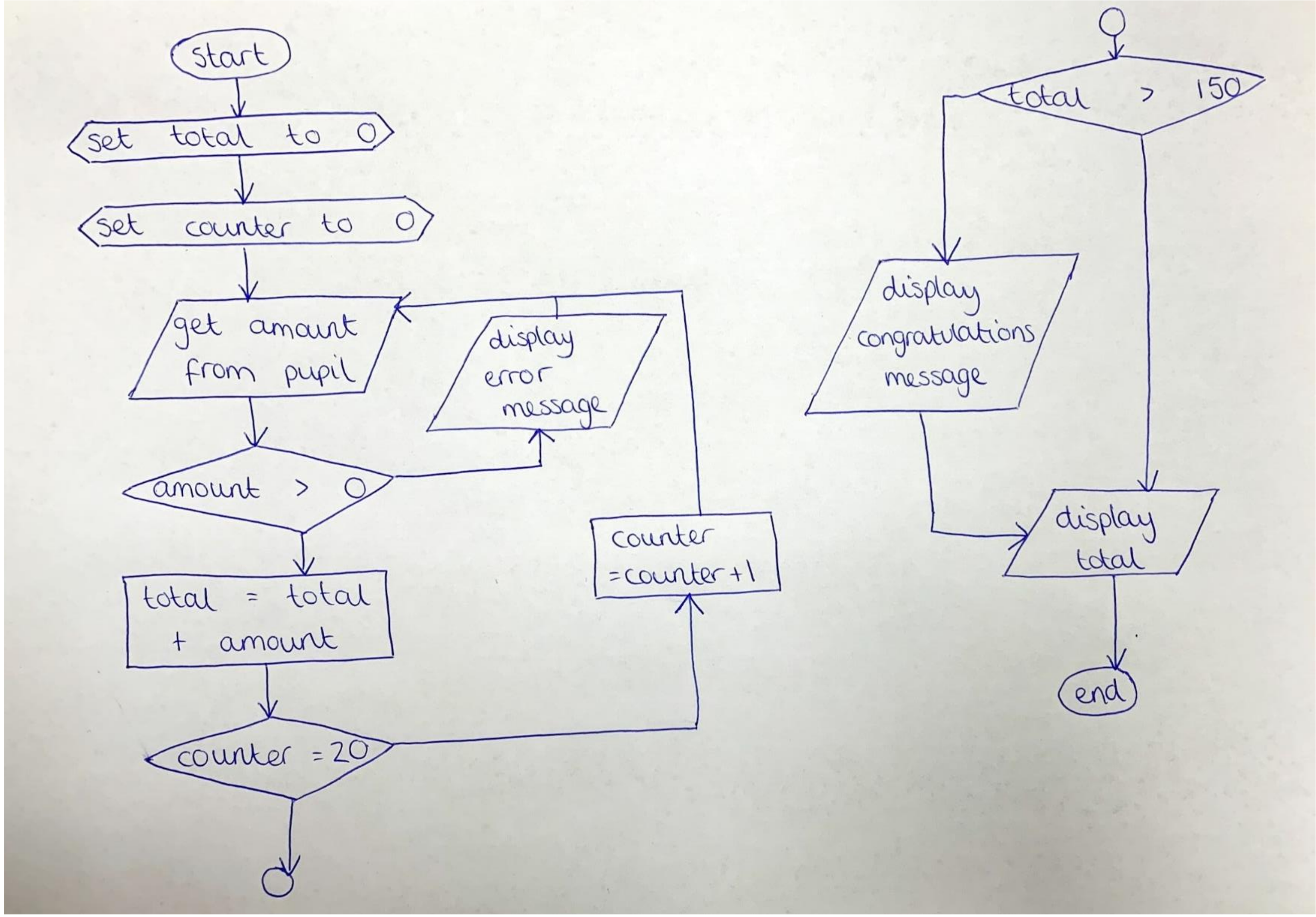


# Flowchart Example

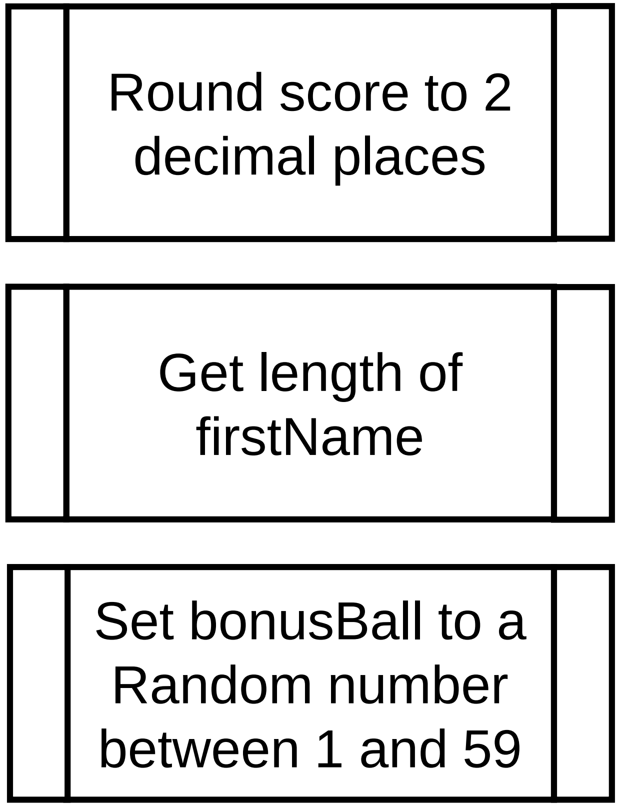
## Program Purpose

A class of 20 pupils is raising money for charity. For each pupil, the program will ask how much money they raised, validate that it is a value larger than 0, and then add the amount to the total. When there are no pupils left, the program should decide if the class has won a prize by raising more than £150, displaying a congratulations message if they have. The program will then display the total amount raised.

# Flowchart Example



# Pre-Defined Functions in Flowcharts



Round score to 2 decimal places

Get length of firstName

Set bonusBall to a Random number between 1 and 59

# Implementation

# Implementation: Data Types and Structures

# Identifying Data Types

There are 5 data types we need to know for National 5:

| Data Type | Types of information stored |
|---|---|
| Character | A single letter or symbol |
| String | Multiple letters or symbols |
| Numeric (Integer) | A positive or negative whole number |
| Numeric (Real) | A positive or negative decimal number |
| Boolean | True or False |

# Using Variables

You should know how to:

- Add information to a variable
- Display information that’s stored in a variable

# Using Variables

Adding information to a variable

```
    name="John Doe"
```

Displaying information stored in a variable

```
    print(name)
```

# Using Arrays

You should know how to:

- Add information to an array
- Display information that’s stored in an array

# Adding information to an array

```
#declaring an array with 5 indices for strings
names = [“”]*5

#setting the value of the array at index 0 and 1
names[0] = “Sam Smith”
names[1] = “John Doe”
```

Note: it is good practice for the name of your array to be plural

You may find different techniques for populating an array online, but this is the technique you should learn for National 5

# Displaying information from an array

```
#Display the contents of the whole array
print(names)
```

Output: [“Sam Smith”, “John Doe”, “”, “”, “”]

```
#Display one item in the array
print[names[0])
```

Output: Sam Smith

# Implementation: Computational Constructs

# Computational Constructs

There are several computational constructs that you need to know about for National 5:

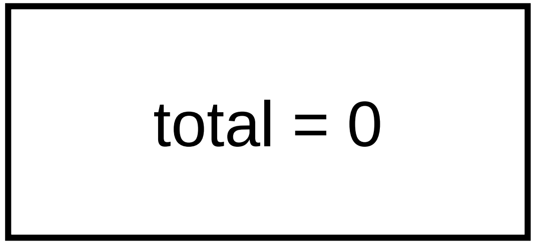
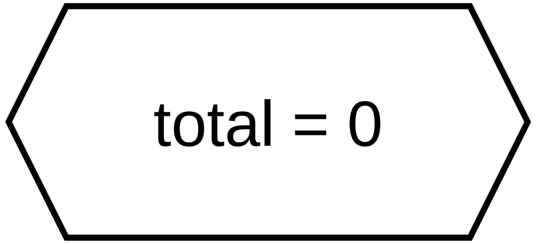
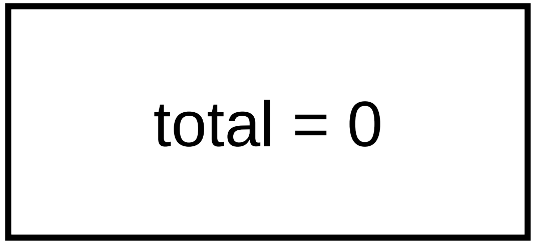
- Assignment
- Basic arithmetic
- Concatenation
- Selection (simple and complex)
- Loops (fixed and conditional)
- Comparison operators
- Logic operators
- Pre-defined functions

# Assignment

Assignment is the process of assigning a value to a variable, or in other words setting a variable to a certain value.

This can be done at the start of a program when declaring a variable (initialisation) or at any time throughout a program.

# What does assignment look like?

| | |
|---|---|
| Pseudocode | SET total to 0 |
| Structure Diagram |  total = 0 |
| Flowchart |  total = 0<br>or<br> total = 0 |
| Python | total = 0 |

# Basic arithmetic

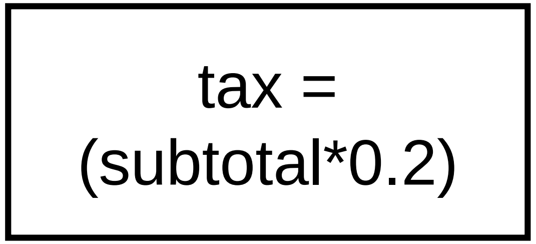
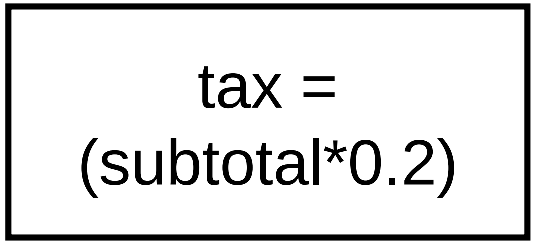
To perform calculations, we need to use arithmetic operators

| Arithmetic | Syntax |
|---|---|
| Add | + |
| Subtract | - |
| Divide | / |
| Multiply | * |
| Exponent (to the power of) | ** |
| Assignment (equals) | = |

# Basic arithmetic

Remember that calculations in your programs follow the principles of BODMAS, so make sure you use brackets to ensure your calculations are executed in the correct order!

# What does basic arithmetic look like?

| | |
|---|---|
| Pseudocode | SET tax TO (subtotal * 0.2) |
| Structure Diagram | <br>tax =<br>(subtotal*0.2) |
| Flowchart | <br>tax =<br>(subtotal*0.2) |
| Python | tax = (subtotal*0.2) |

# Concatenation

Concatenation is the process of joining strings together to make a new string.

# What does concatenation look like?

| | |
|---|---|
| Pseudocode | username = “13517” & surname & firstName |
| Structure Diagram | username = “13517” & surname & firstName |
| Flowchart | username = “13517” & surname & firstName |
| Python | username = “13517” + surname + firstName |

# Comparison operations

We often need to compare values to find out if they are equal, not equal, or to find out which one is bigger or smaller.

# Comparison operators

| Comparison | Syntax |
|---|---|
| Is a equal to b | a == b |
| Is a not equal to b | a != b or a <> b or a ≠ b |
| Is a less than b | a < b |
| Is a greater than b | a > b |
| Is a less than or equal to b | a <= b or a ≤ b |
| Is a greater than or equal to b | a >= b or a ≥ b |

# Comparison operations

Using comparison operators will return either True or False. For example, if variable a is 9 and variable b is 19

a > b would return false

a != b would return true

a < b would return true

We use comparison operators in selection statements. If the comparison is True then the program will execute the code inside the selection statement.

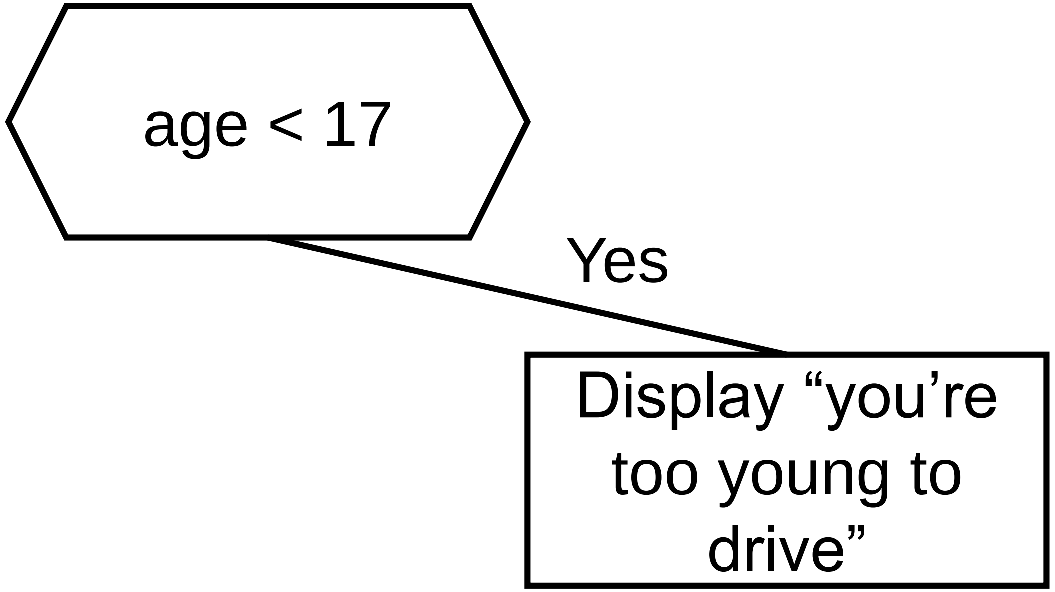
# Simple selection statements

Programs need to be able to make decisions. IF statements allow a program to select an action depending on the value of specific variables.

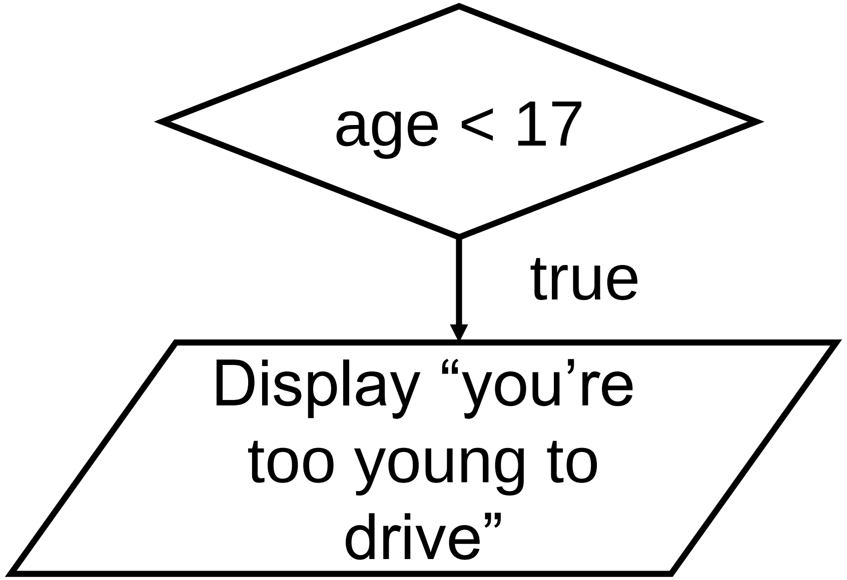
IF statements make their decisions using a condition that can either be `True` or `False`.

If the condition is met then the program will execute a piece of code. If the condition is not met, the program will not execute the piece of code.

# What does a simple selection (if statement) look like?

| | |
|---|---|
| Pseudocode | IF age < 17 THEN<br>    SEND “you're too young to drive” TO DISPLAY<br>END IF |
| Structure Diagram | <br>age < 17<br>Yes<br>Display “you’re too young to drive” |

# What does a simple selection (if statement) look like?

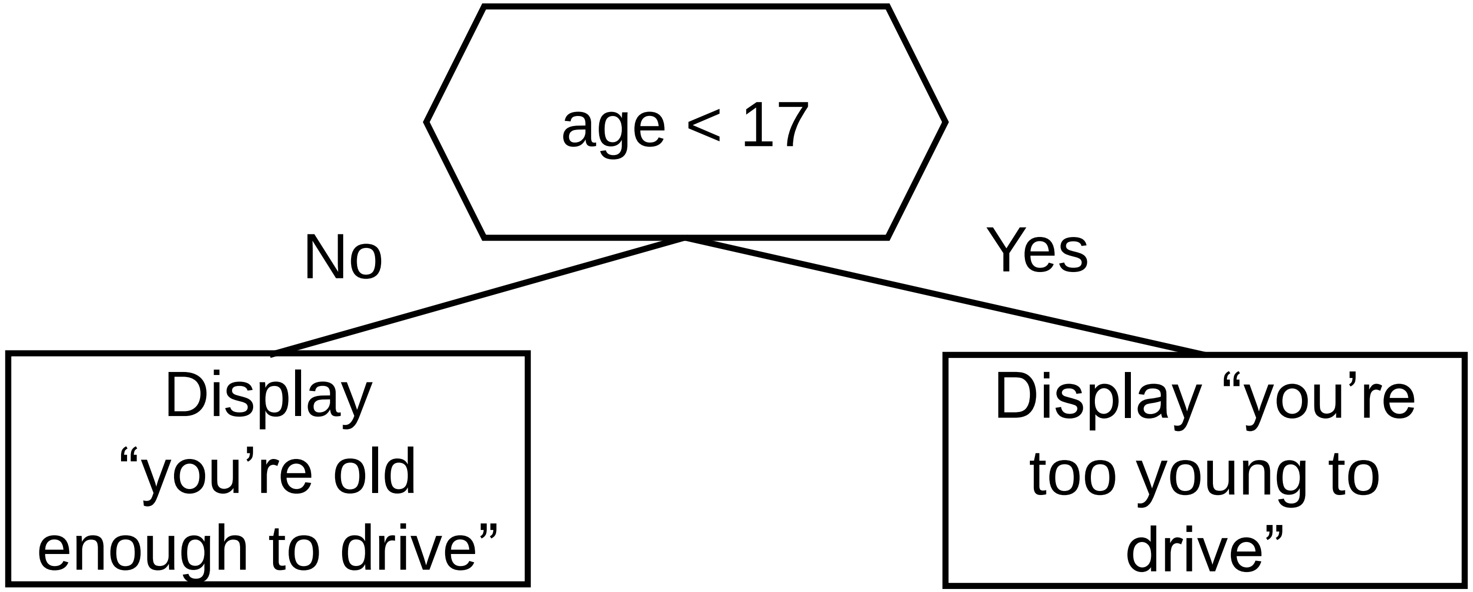
| | |
|---|---|
| Flowchart |  |
| Python | if age < 17:<br>    print (“you’re too young to drive”) |

# Simple selection statements with else

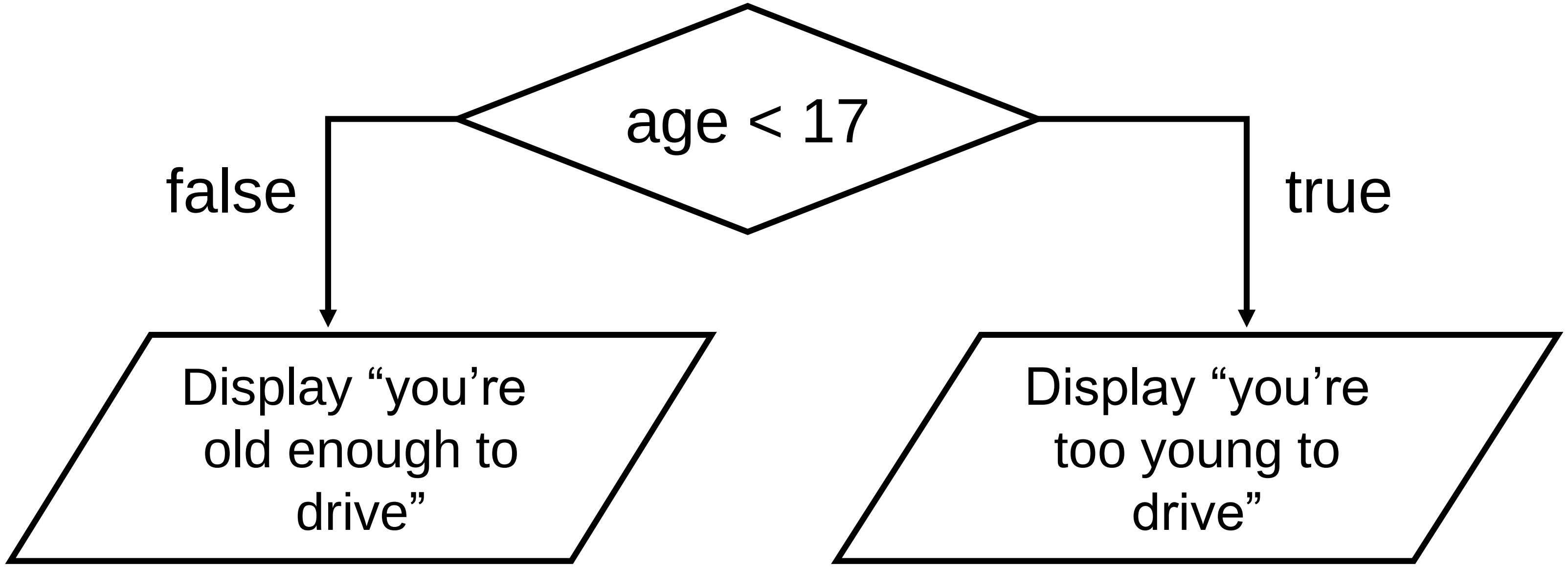
Sometimes we may want our programs to execute different pieces of code under different circumstances.

We can use IF, ELSE statements if we want to do one thing if the condition is met, and a different thing if the condition is not met.

# What does a simple selection (if, else statement) look like?

| | |
|---|---|
| Pseudocode | IF age < 17 THEN<br>    SEND “you're too young to drive” TO DISPLAY<br>ELSE<br>    SEND “you’re old enough to drive” TO DISPLAY<br>END IF |
| Structure Diagram | <br>age < 17<br>No<br>Yes<br>Display “you’re old enough to drive”<br>Display “you’re too young to drive” |

# What does a simple selection (if, else statement) look like?

| | |
|---|---|
| Flowchart |  age < 17<br>false<br>true<br>Display “you’re old enough to drive”<br>Display “you’re too young to drive” |
| Python | if age < 17:<br>    print (“you’re too young to drive”)<br>else:<br>    print (“you’re old enough to drive”) |

# Simple selection statements with multiple branches

We can also use if statements to test multiple conditions, and execute different pieces of code if each condition is met.

We achieve this using IF, ELSE IF, ELSE statements.

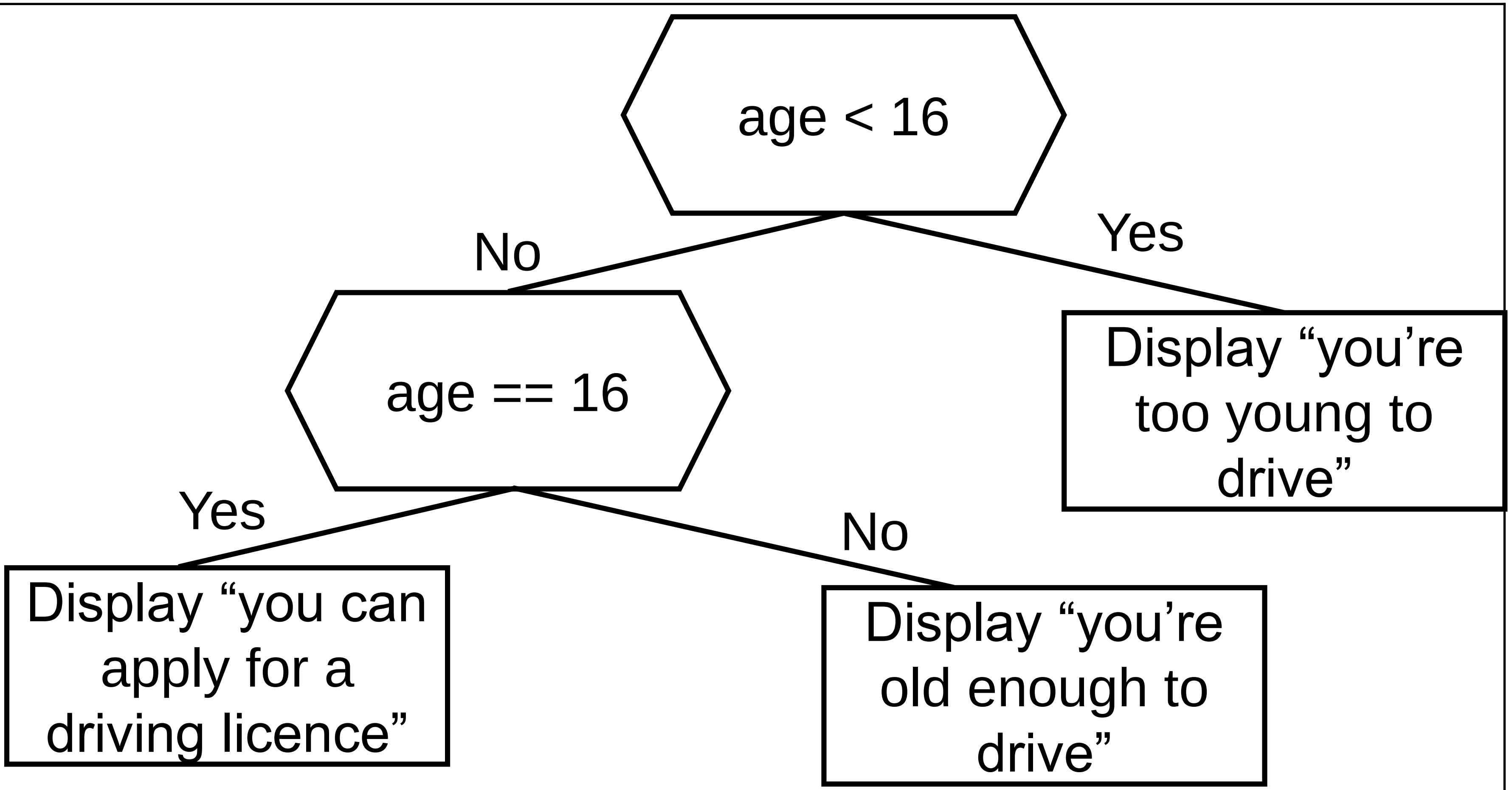
# What does a simple selection (if, else if, else statement) look like?

Pseudocode

```
IF age < 16 THEN
    SEND “you're too young to drive” TO DISPLAY
ELSE IF age == 16 THEN
    SEND “you can apply for a driving licence” TO DISPLAY
ELSE
    SEND “you’re old enough to drive”
END IF
```

# What does a simple selection t (if, else if, else statement) look like?

Structure Diagram



age < 16

No

Yes

age == 16

Display “you’re too young to drive”

Yes

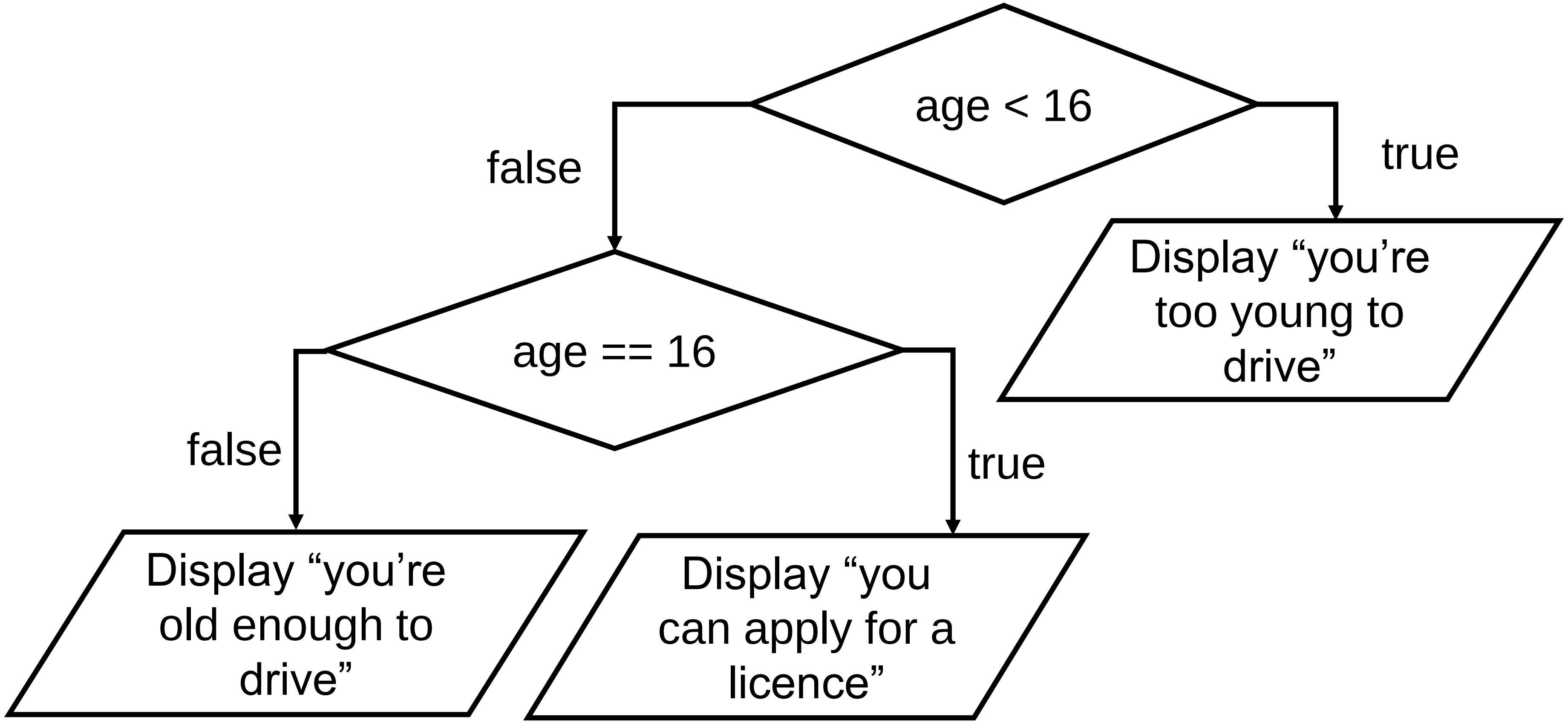
No

Display “you can apply for a driving licence”

Display “you’re old enough to drive”

# What does a simple selection (if, else if, else statement) look like?

Flowchart


age < 16
false
true
Display "you're too young to drive"
age == 16
false
true
Display "you're old enough to drive"
Display "you can apply for a licence"

# What does a simple selection (if, else if, else statement) look like?

| | |
|---|---|
| Python | if age < 16:<br>    print (“you’re too young to drive”)<br>elif age == 16:<br>    print (“you can apply for a driving licence”)<br>else:<br>    print (“you’re old enough to drive”) |

# Logic operations

Logic operations are used to build more complex expressions, meaning we can check the value of two or more variables at the same time.

# Logic operators

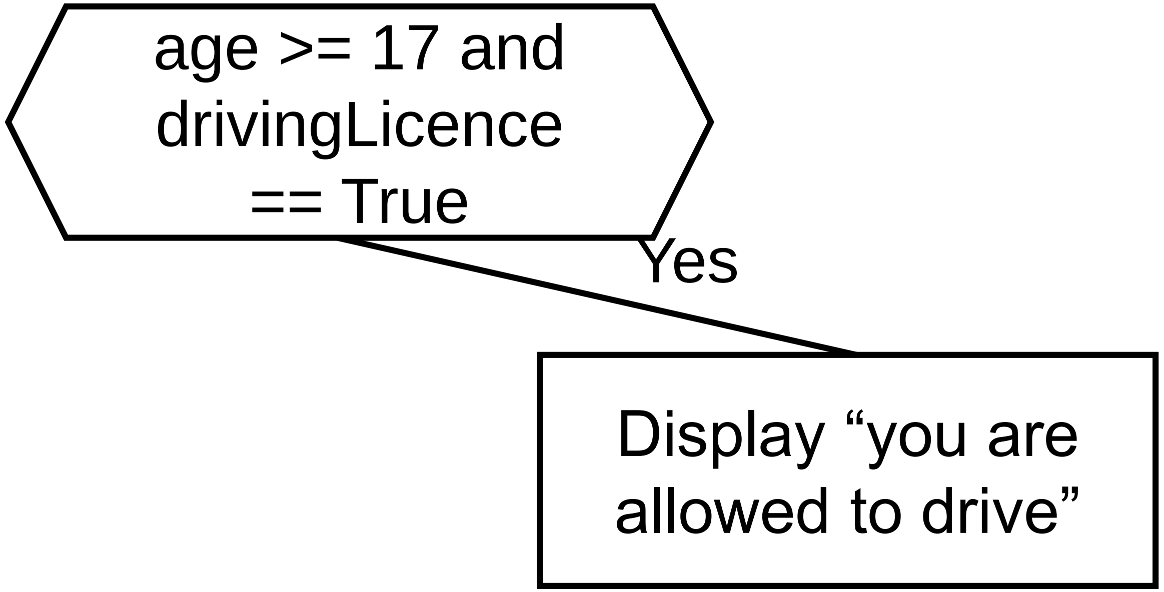
| Operator | Example | Explanation |
|---|---|---|
| AND | score > 0 AND score < 10 | The expression on both sides of the AND operator must be true for AND to return TRUE. If either is FALSE the overall result is FALSE. |
| OR | age = 16 OR age = 17 | If the expression on either side of the OR operator is TRUE then the overall result is TRUE. (This also returns TRUE if both sides are TRUE). |
| NOT | NOT (year == “S4”) | This operator inverts the result of the expression following it - if the expression after the NOT operator is TRUE, the result will be FALSE. If the expression after the NOT operator is FALSE, the result will be TRUE |

# Complex selection statements

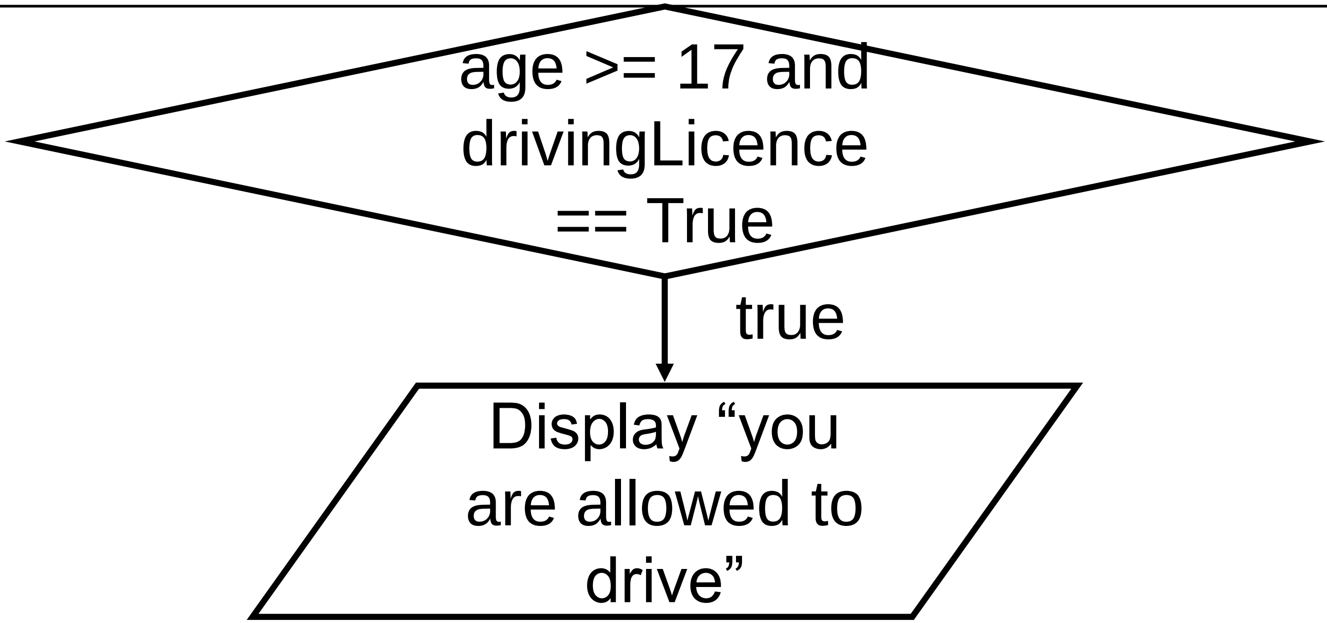
We use logic operators in selection statements to build complex conditions. If the result of the whole condition is True then the program will execute the code inside the selection statement.

| Operator | |
|---|---|
| AND | Only executes if both conditions are true |
| OR | Executes if any of the conditions are true |
| NOT | Executes if the condition is false, as the NOT will invert this and change it to true. |

# What does a complex selection (if statement) look like?

| | |
|---|---|
| Pseudocode | IF age >= 17 AND drivingLicence == True THEN<br>    SEND “you are allowed to drive” TO DISPLAY<br>END IF |
| Structure Diagram |   age >= 17 and drivingLicence == True<br>Yes<br>Display “you are allowed to drive” |

# What does a complex selection (if statement) look like?

| | |
|---|---|
| Flowchart |  age >= 17 and drivingLicence == True<br>true<br>Display “you are allowed to drive” |
| Python | if age >= 17 and drivingLicence == True:<br>    print (“you are allowed to drive”) |

# Iteration

Sometimes code in a program needs to be run multiple times.
This is called iteration, or repetition.

In programming, this is achieved using loops.

There are two types of loops

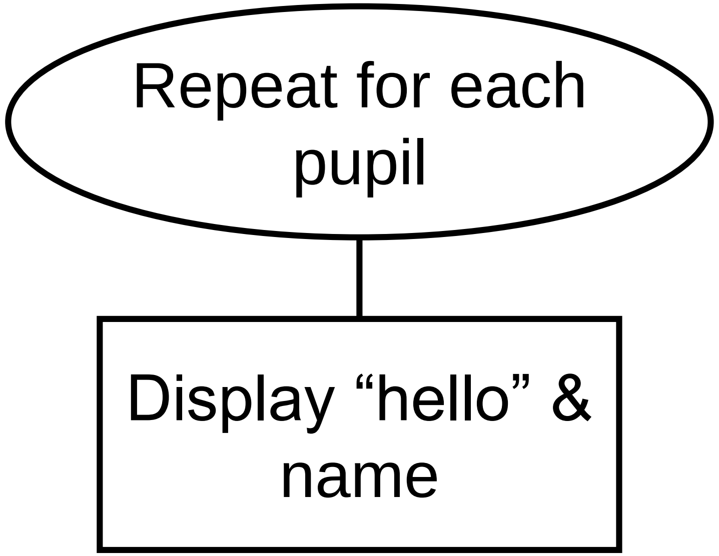
- Fixed loops
- Conditional loops

# Fixed loops

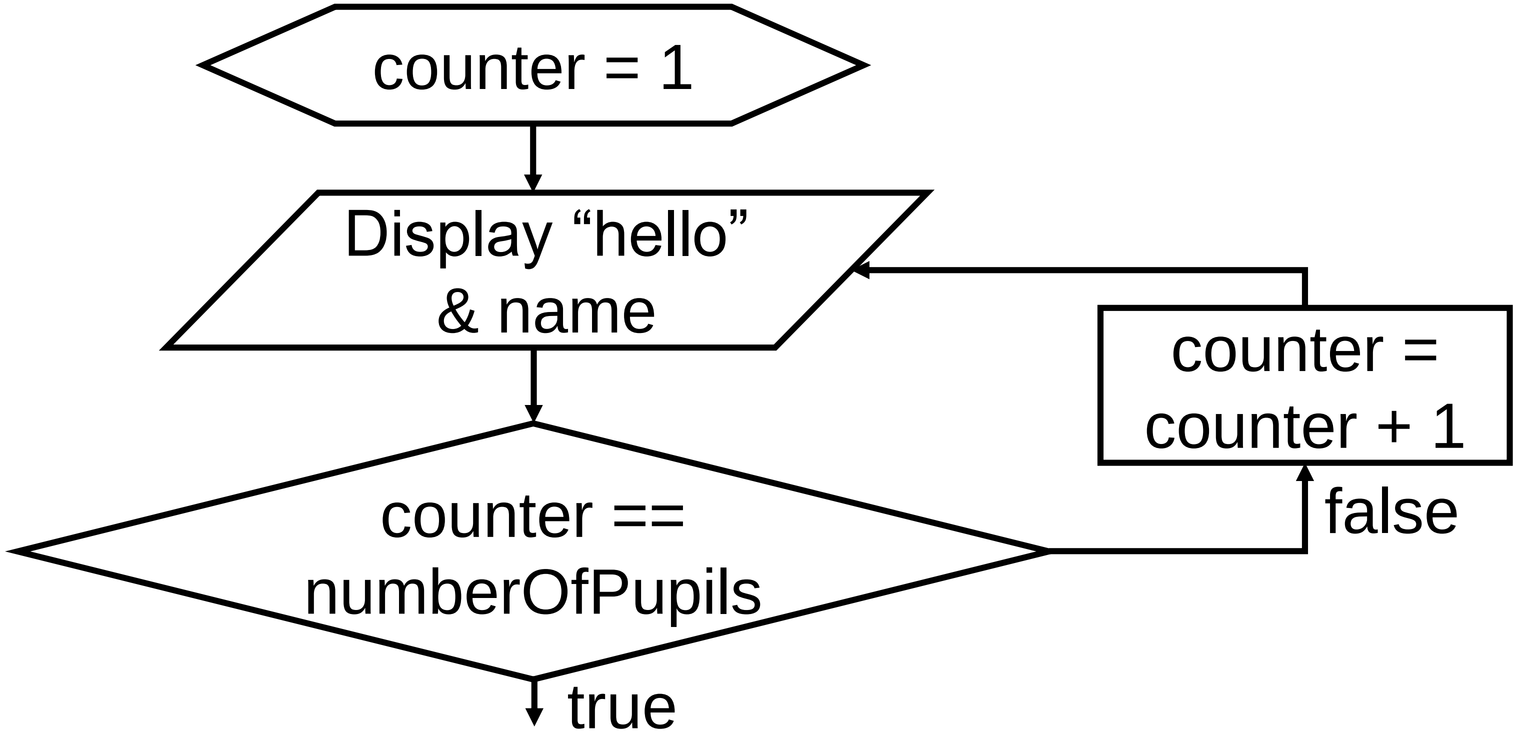
A fixed loop is used when we have an idea about how many times the code will have to repeat.

You can hardcode how many times a fixed loop repeats, or you can use a variable in your program that it set by the user before the loop is executed.

# What does a fixed loop look like?

| | |
|---|---|
| Pseudocode | FOR loop = 1 to numberOfPupils<br>    SEND “hello ” & name TO DISPLAY<br>END LOOP |
| Structure Diagram | <br>Repeat for each pupil<br>Display “hello” & name |

# What does a fixed loop look like?

<table>
<tr><td>Flowchart</td><td>
</td></tr>
<tr><td>Python</td><td>

```
for counter in range (0, 10):
     print (“hello ” + name)
```

</td></tr>
</table>

# Conditional loops

A conditional loop is used when we need to repeat code until a condition is met, for example the program might keep asking a user to enter their password until they enter the right one.

We do not know how many times the code is going to run before we enter the loop.

There are two types of conditional loops:

- Pre-test
- Post-test

# Pre-test loops

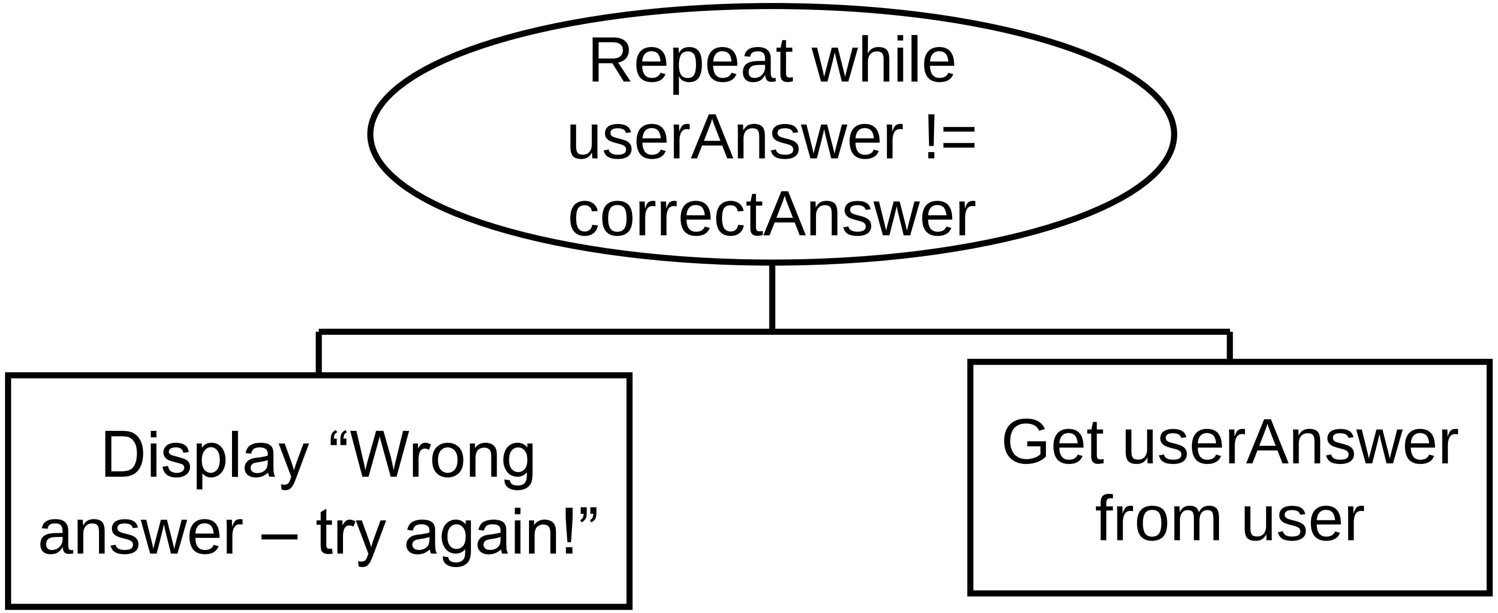
Pre-test loops, or WHILE loops, test a condition at the very start of the loop. If the condition is not met, then the code inside the loop never executes.

# Post-test loops

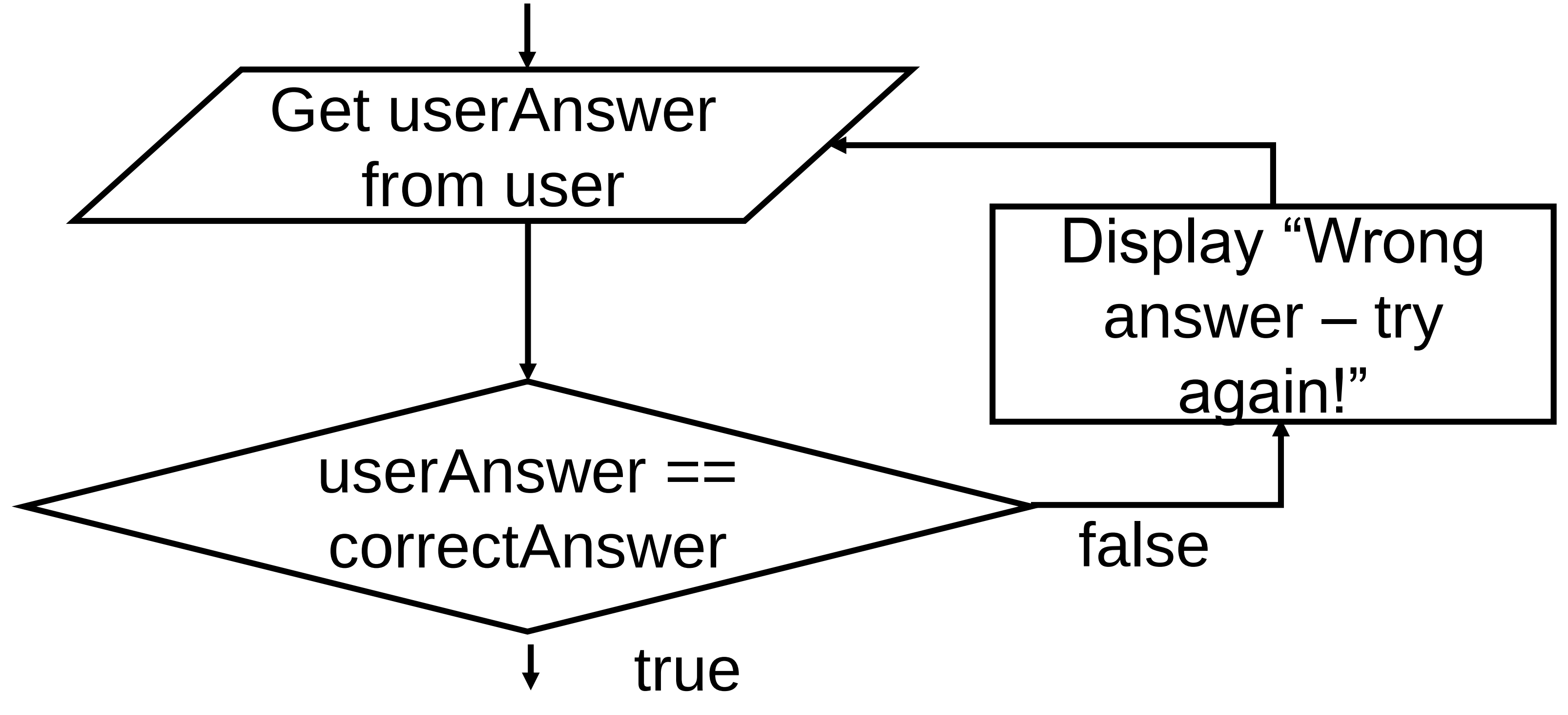
Post-test loops, or DO UNTIL loops, test a condition at the end of the loop. The code inside the loop executes at least once.

Python only has one type of loop, which is why we haven’t looked at post-test loops in class, however you may come across these loops in exam papers.

# What does a conditional loop look like?

| | |
|---|---|
| Pseudocode | WHILE userAnswer != correctAnswer<br>    SEND “Wrong answer – try again!” TO DISPLAY<br>    RECEIVE userAnswer FROM KEYBOARD<br>END LOOP |
| Structure Diagram | <br>Repeat while userAnswer != correctAnswer<br>Display “Wrong answer – try again!”<br>Get userAnswer from user |

# What does a conditional loop look like?

| | |
|---|---|
| Flowchart | <br>Get userAnswer from user<br>userAnswer == correctAnswer<br>false<br>Display “Wrong answer – try again!”<br>true |
| Python | while userAnswer != correctAnswer:<br>    print (“Wrong answer – try again!”)<br>    userAnswer = input() |

# Pre-Defined Functions

A pre-defined function is some code that has already been written and is ready for us to use (call).

Pre-defined functions are usually small tasks that are often carried our by programmers. These functions have usually been made as efficient as possible.

There are 3 pre-defined functions that we need to know for National 5.

- Random
- Round
- Length

# Pre-Defined Functions

Random is used to generate a random number

Round is used to round number to a certain number of decimal places

Length is used to find out the length of a string (we can also use this pre-defined function with arrays when we look at those)

# Pre-defined Functions

To call (use) a defined function we use brackets, e.g. round(), randint() or len().

We also have to pass information in to these functions so that the program knows what we want it to do. Passing information into functions is done using parameters.

# round()

The round() function is used to round a number to a certain number of decimal places.

There are two pieces of information we need to give to this function as parameters.

Parameter 1: what number do you want to round

Parameter 2: How many decimal places do you want to round to?

# round() Example

The round() function is used to round a number to a certain number of decimal places.

```
x = round(5.76234, 2)
print(x)
```

This would output **5.76**

# round() Example

We can also pass in variables as parameters.

```
pi = 3.141592653589793
piRounded = round(pi, 2)
print(piRounded)
```

This would output **3.14**

# randint()

randint() is used to return a random number between the given range.

There are two pieces of information we need to give to this function as parameters.

Parameter 1: the lower limit

Parameter 2: the upper limit

# randint() Example

randint() is used to return a random number between the given range. Random functionality is not included in Python as standard, so we have to import this code.

```
import random
print(random.randint(3, 9))
```

This could output **3**, **4**, **5**, **6**, **7**, **8** or **9**

# len()

len() is used to find the length of the string or array.

There is one piece of information we need to give to this function as a parameter.

Parameter 1: the string you want to know the length of

# len() Example

len() is used to find the length of the string.

```
print(len(“Hello World!”))
```

This would output 12 – remember you need to include any punctuation or spaces.

# len() Example

We can also pass in variables as parameters.

```
string = "geeks"
print(len(string))
```

This would output 5

# len() Example: Arrays

We can use the same pre-defined function to determine the length of an array – in fact you can picture a string as an array of characters.

```
names = [“”]*5

lengthOfArray = len(names)
print(lengthOfArray)
```

This would output 5

# Implementation: Standard Algorithms

# Standard Algorithms

There are three standard algorithms you need to know for National 5:

- Running Total
- Input Validation
- Traversing a 1D Array

# Running Total

The Running Total standard algorithm uses a fixed or conditional loop, taking in values from a user and adding them together.

There are 4 steps to a running total algorithm:

1. Set total variable to 0
2. Start loop
3. Ask the user to enter a number
4. Add the number to the total

# Running Total: Pseudocode

With a fixed loop

```
SET total TO 0
FOR loop = 1 to 20
    RECEIVE amountRaised FROM KEYBOARD
    total = total + amountRaised
END LOOP
SEND total TO DISPLAY
```

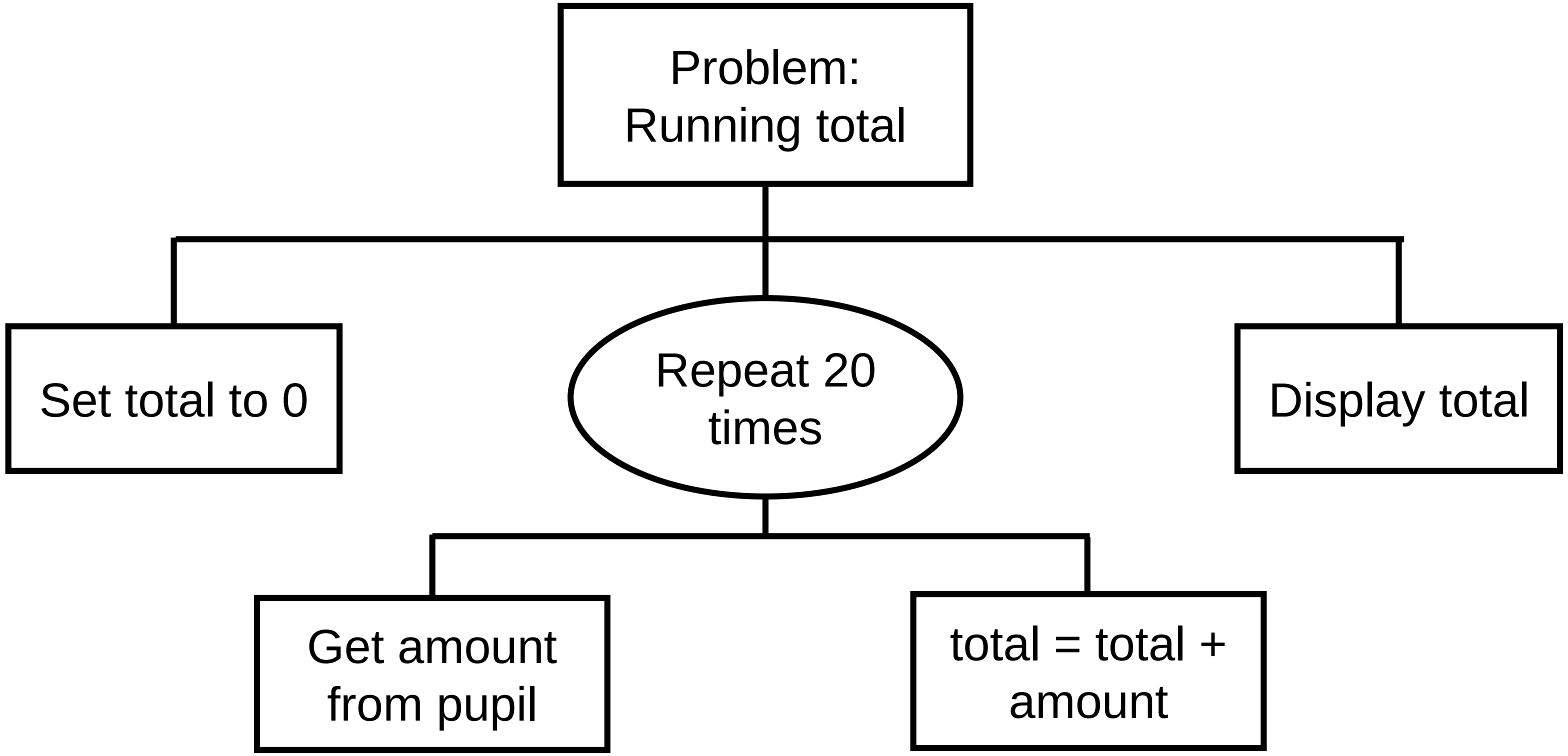
# Running Total: Pseudocode

With a conditional loop

```
SET total TO 0
SET morePupils TO True
WHILE morePupils = True
    RECEIVE amountRaised FROM KEYBOARD
    total = total + amountRaised
    RECEIVE morePupils FROM KEYBOARD
END LOOP
SEND total TO DISPLAY
```

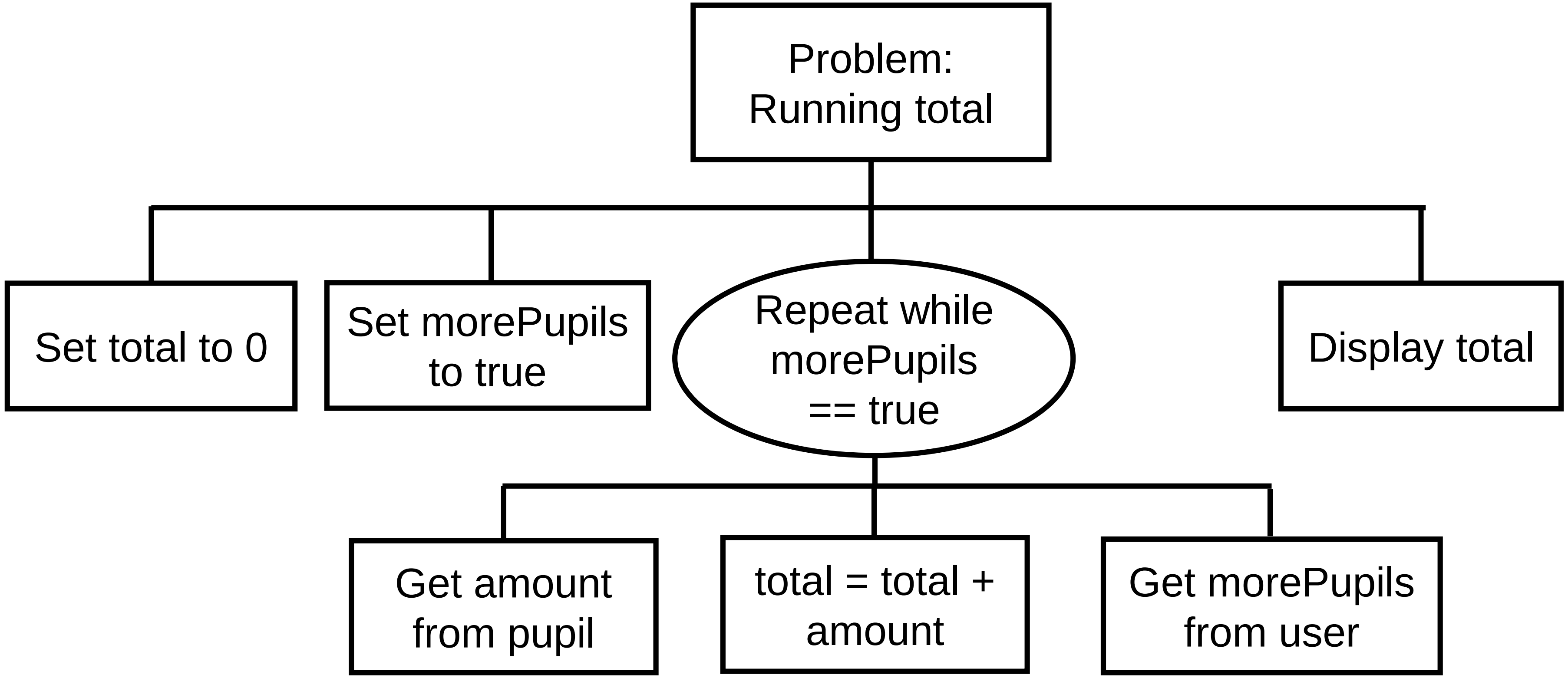
# Running Total: Structure Diagram

With a fixed loop



# Running Total: Structure Diagram

With a conditional loop



Problem: Running total

Set total to 0

Set morePupils to true

Repeat while morePupils == true

Get amount from pupil

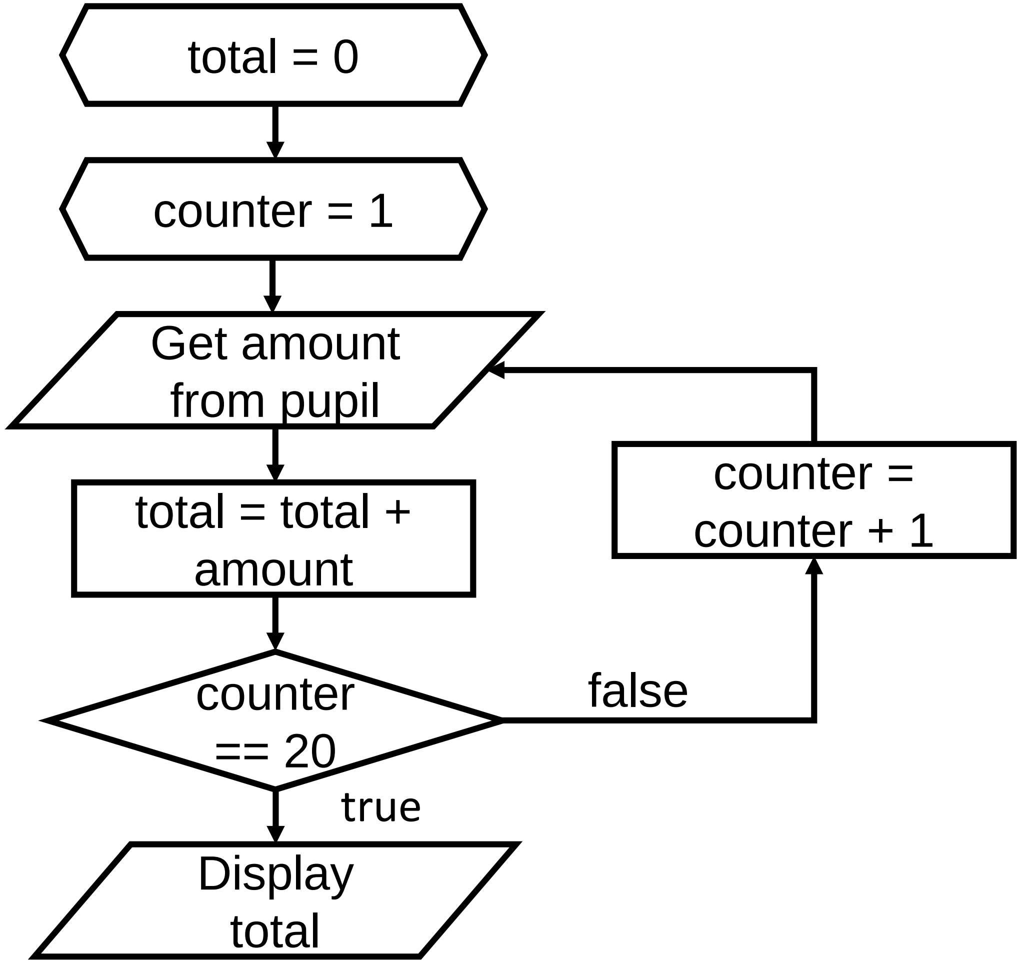
total = total + amount

Get morePupils from user

Display total

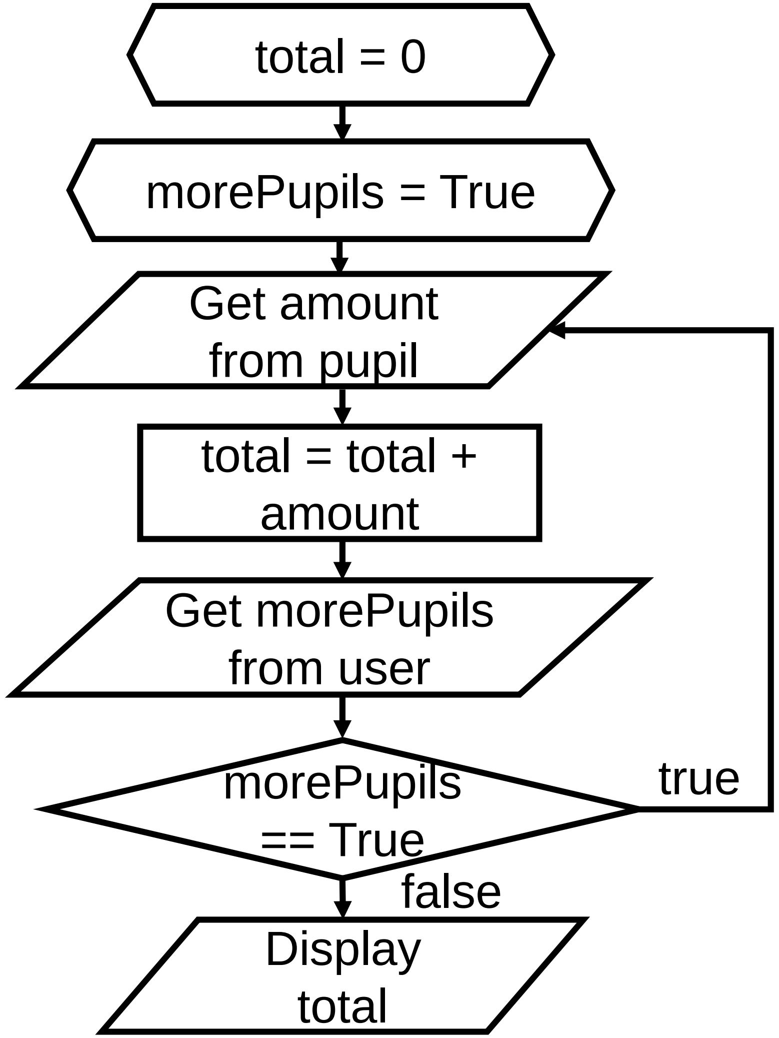
# Running Total: Flowchart

With a fixed loop



# Running Total: Flowchart

With a conditional loop



# Running Total: Python

## With a fixed loop

```
total = 0

for counter in range (0, 20):
   amount = int(input(“Enter amount raised”))
   total = total + amount

print(total)
```

# Running Total: Python

With a conditional loop

```
total = 0
morePupils = “Y”

while morePupils == “Y”:
   amount = int(input(“Enter amount raised”))
   total = total + amount
   morePupils = input(“Are there more pupils? Y/N”)

print(total)
```

# Input Validation

The Input Validation standard algorithm uses a conditional loop, and continues to ask the user for input until an acceptable answer is provided.

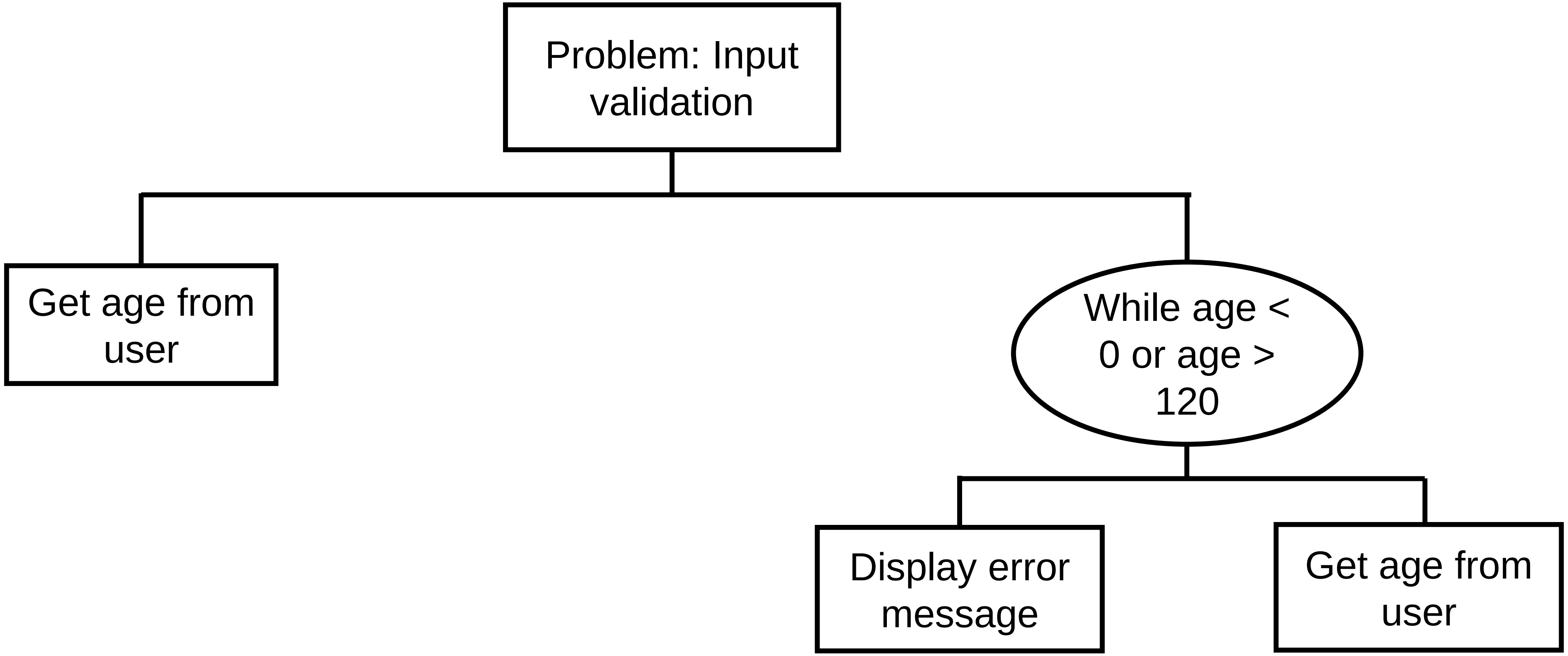
There are 4 steps to an input validation algorithm:

1. Receive value from user
2. Start conditional loop **if value is unacceptable**
3. Display an error message
4. Receive value from user

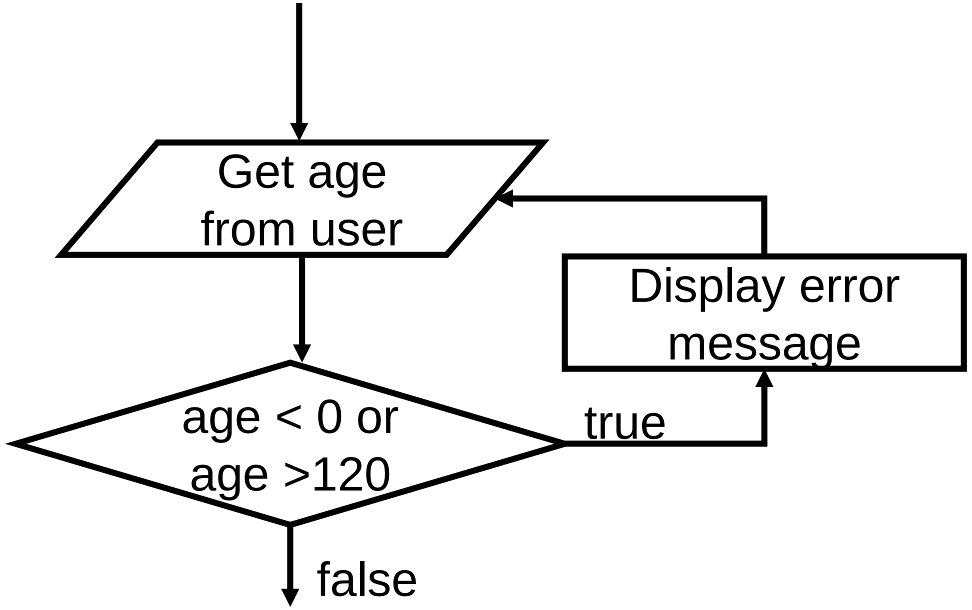
# Input Validation: Pseudocode

```
RECEIVE age FROM KEYBOARD
WHILE age < 0 OR age > 120
    SEND “Invalid age.  Enter age between 0 and 99” TO DISPLAY
    RECEIVE age FROM KEYBOARD
END LOOP
```

# Input Validation: Structure Diagram



# Input Validation: Flowchart



# Input Validation: Python

```
age = int(input(“Enter your age: ”)

while age < 0 or age > 120:
   print(“Invalid age – enter 0-99 only”)
   age = int(input(“Enter your age: ”)
```

# Traversing a 1D Array

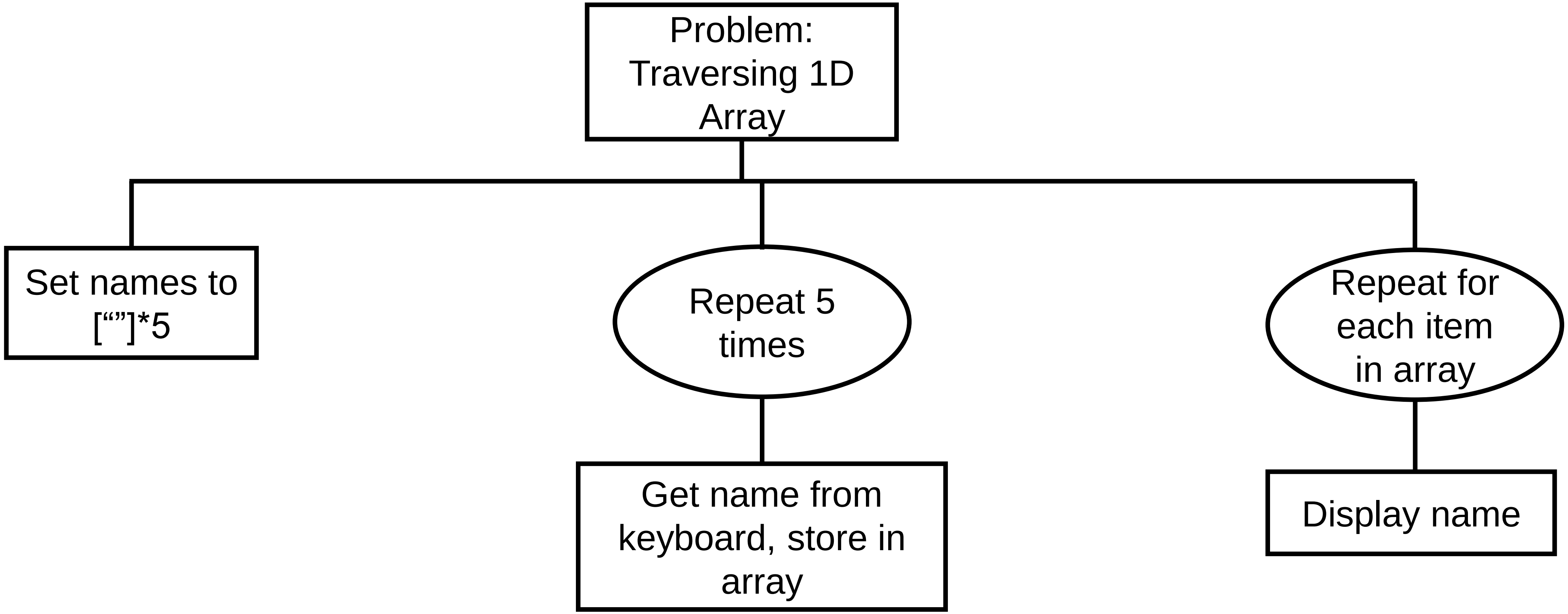
We can use the index of an array to move through the array within a fixed loop. This is called traversing the array.

This standard algorithm is usually used to add values to an array, or to display every item in an array.

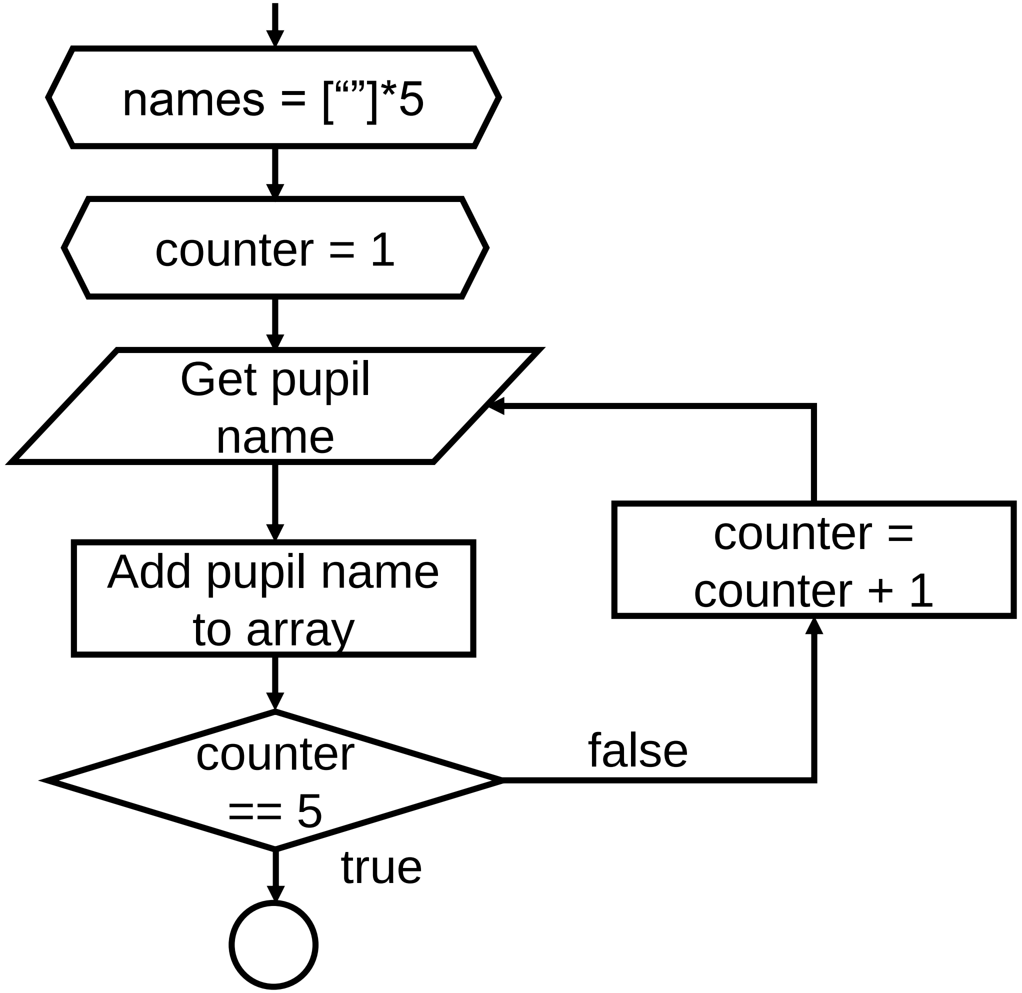
# Traversing a 1D Array: Pseudocode

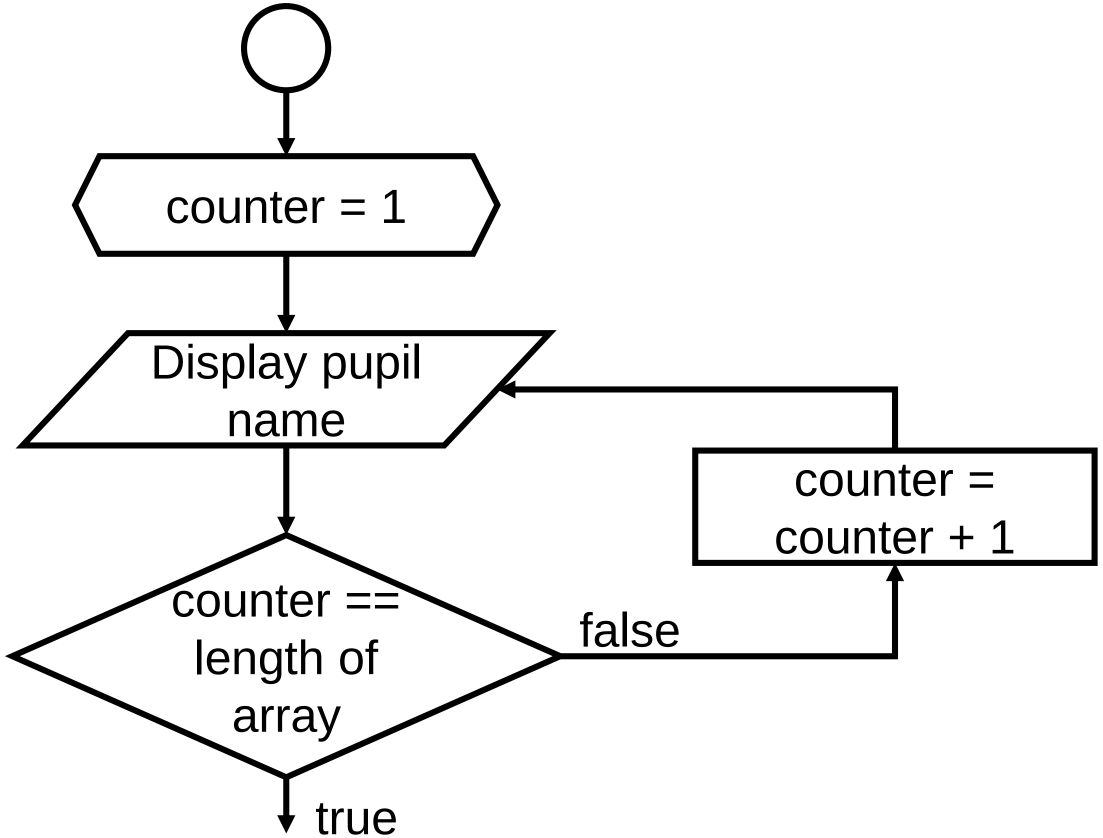
```
SET names TO array of strings
FOR loop = 1 to 5
    RECEIVE name FROM KEYBOARD
    STORE name in array
END LOOP
FOR each item in the array
    SEND name TO DISPLAY
END LOOP
```

# Traversing a 1D Array: Structure Diagram


Problem: Traversing 1D Array
Set names to [“”]*5
Repeat 5 times
Get name from keyboard, store in array
Repeat for each item in array
Display name

# Traversing a 1D Array: Flowchart





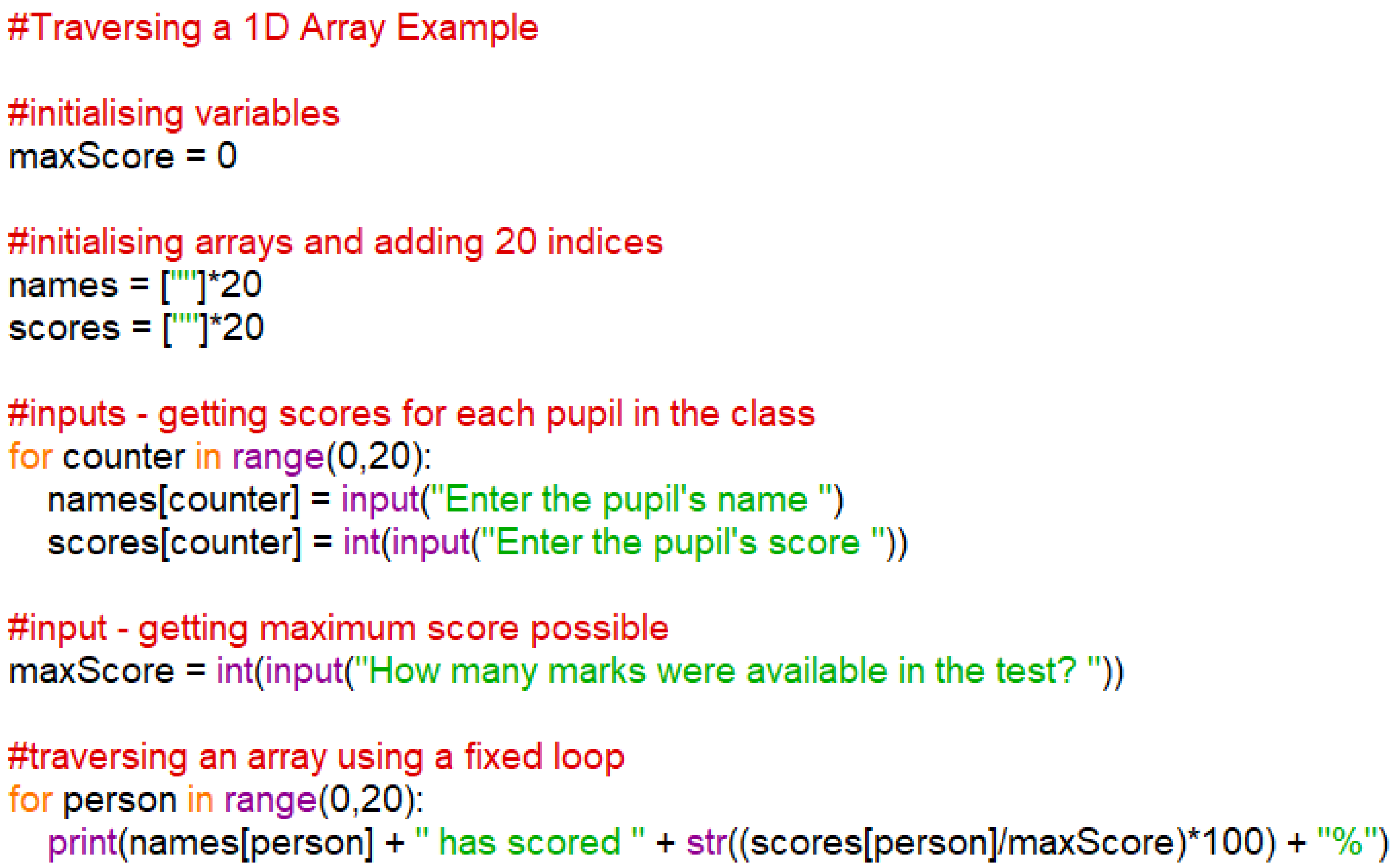
# Traversing a 1D Array: Python

```
names = [“”]*5

for counter in range(0, 5):
   names[counter] = input(“Enter the pupil’s name.”)

for counter in range(0, len(names)):
   print(“Pupil name: ” + names[counter])
```

# Traversing a 1D Array: Worked Example



```
#Traversing a 1D Array Example

#initialising variables
maxScore = 0

#initialising arrays and adding 20 indices
names = [""]*20
scores = [""]*20

#inputs - getting scores for each pupil in the class
for counter in range(0,20):
    names[counter] = input("Enter the pupil's name ")
    scores[counter] = int(input("Enter the pupil's score "))

#input - getting maximum score possible
maxScore = int(input("How many marks were available in the test? "))

#traversing an array using a fixed loop
for person in range(0,20):
    print(names[person] + " has scored " + str((scores[person]/maxScore)*100) + "%")
```

# Testing

# Testing

Testing is the process of making sure that the program works as intended – does the program match the design?

There are two areas we consider when testing software:

- Test data
- Types of errors

# Test tables

Programs are tested to make sure they work as we expect.

This is achieved by creating test tables which contain different test data. The test table contains the expected output for each type of test data. As the program is tested, the actual output is recorded in the table.

If the expected output matches the actual output then the program works as expected.

# Test data

There are 3 types of test data:

Normal: data which the program should accept and which is within the expected range for the program.

Extreme: data that is on the boundaries of what should be acceptable, or on the boundaries of conditions within the program.

Exceptional: data that would not be accepted by the program, and should be rejected. Designed to test whether or not the program can cope with unexpected data.

# Test table example

Here is a testing table we would use to test a program which asks a school pupil to enter their age and displays if they should be in primary or secondary.

| Test Type | Test Data | Expected Output | Actual Output |
|---|---|---|---|
| Normal | 7 | Primary | |
| Extreme | 18 | Secondary | |
| Exceptional | 21 | Error message | |

During testing, the actual output column would be filled out. If the actual output matches the expected output then the program works as expected.

# Types of errors

There are 3 types of errors:

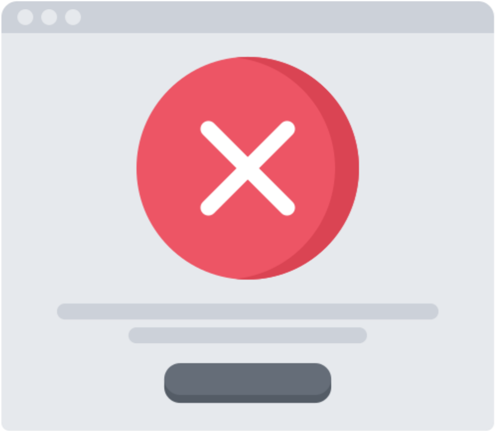
- Syntax errors
- Logic errors
- Run-time errors (or execution errors)

You need to be able to identify and fix each type of error.

# Syntax errors

Syntax errors are usually typing or spelling mistakes in the program.

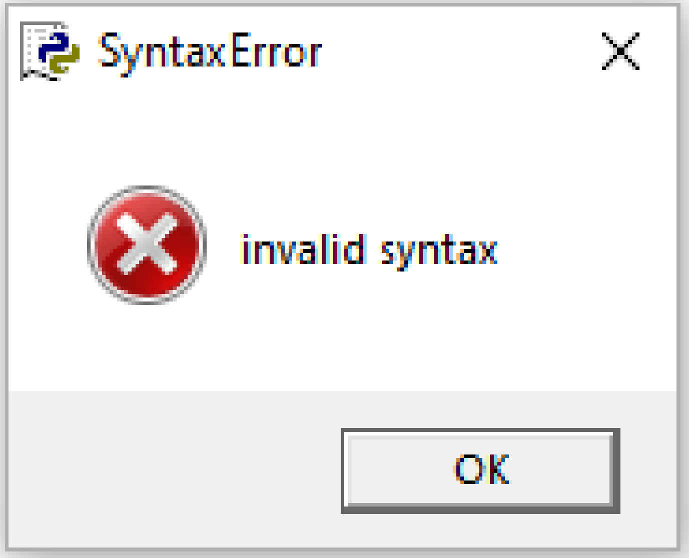
- Misspelling a reserved word
- Misspelling a variable name
- Missing brackets or colons



A reserved word is one that would turn purple or orange in Python e.g. print or while

# Finding syntax errors

Syntax errors are found before the program executes – the development environment will not allow the program to run and will display an error.



SyntaxError

invalid syntax

# Logic errors

A logic error happens when the program is syntactically correct, but there is a mistake in the way a calculation or condition has been written.

- Using < instead of >
- Using and instead of or
- Forgetting to use brackets in calculations

# Finding logic errors

Logic errors are found during the testing process. A test table is written with the expected output, but when the program is actually run it produces a different result.

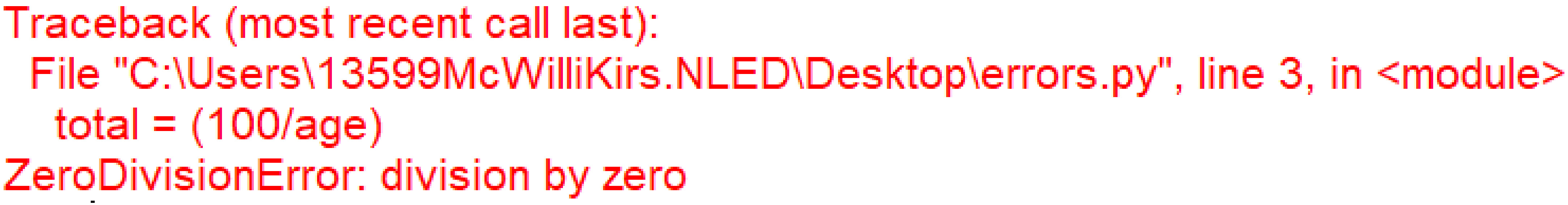
# Run-time errors

Run-time errors (sometimes called execution errors) happens when something unexpected happens while the program is executing.

- Trying to divide by zero
- Trying to complete a calculation with a string
- Trying to access a file that doesn’t exist

# Finding run-time errors

Run-time errors are found while the program is running. If something unexpected happens the program will crash and display a red error message.



```
Traceback (most recent call last):
  File "C:\Users\13599McWilliKirs.NLED\Desktop\errors.py", line 3, in <module>
    total = (100/age)
ZeroDivisionError: division by zero
```

# Evaluation

# Evaluation

When evaluating a piece of software we consider several areas:

- Fitness for purpose
- Efficiency
- Robustness
- Readability

# Fitness for Purpose

Fitness for purpose answers the question “does your software meet the requirements identified during the Analysis stage of the software development lifecycle.

If you have a requirement to display information to the user, and your program doesn’t do that, your program is not fit for purpose.

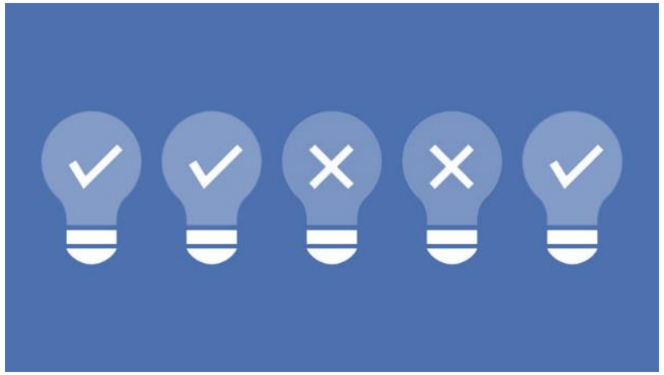
# Efficiency

Efficiency is evaluated in two areas:

- How much processing power is required to run the code?
- How much time does it take the developer to write the code?

There are three constructs that we've looked at that make code more efficient.

- If statements
- Arrays
- Fixed loops



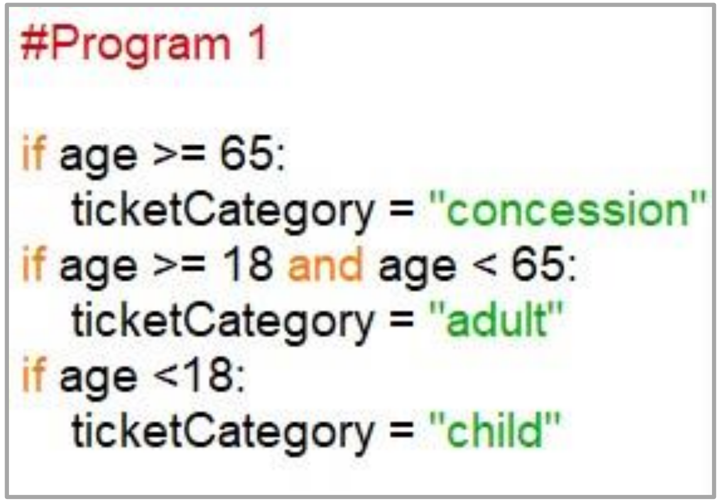
# Efficiency: If Statements

Comparing these programs, we can see that Program 2 is more efficient.

Program 1 will **always** evaluate all three if conditions, even if the first one is evaluated to true.

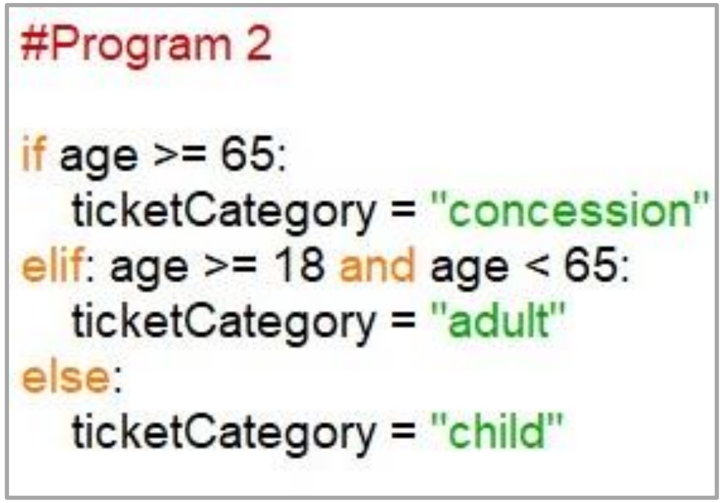
Program 2 will not evaluate further conditions if the first condition is evaluated to true.

This reduces the number of instructions that need to be processed by the ALU.



```
#Program 1

if age >= 65:
    ticketCategory = "concession"
if age >= 18 and age < 65:
    ticketCategory = "adult"
if age <18:
    ticketCategory = "child"
```



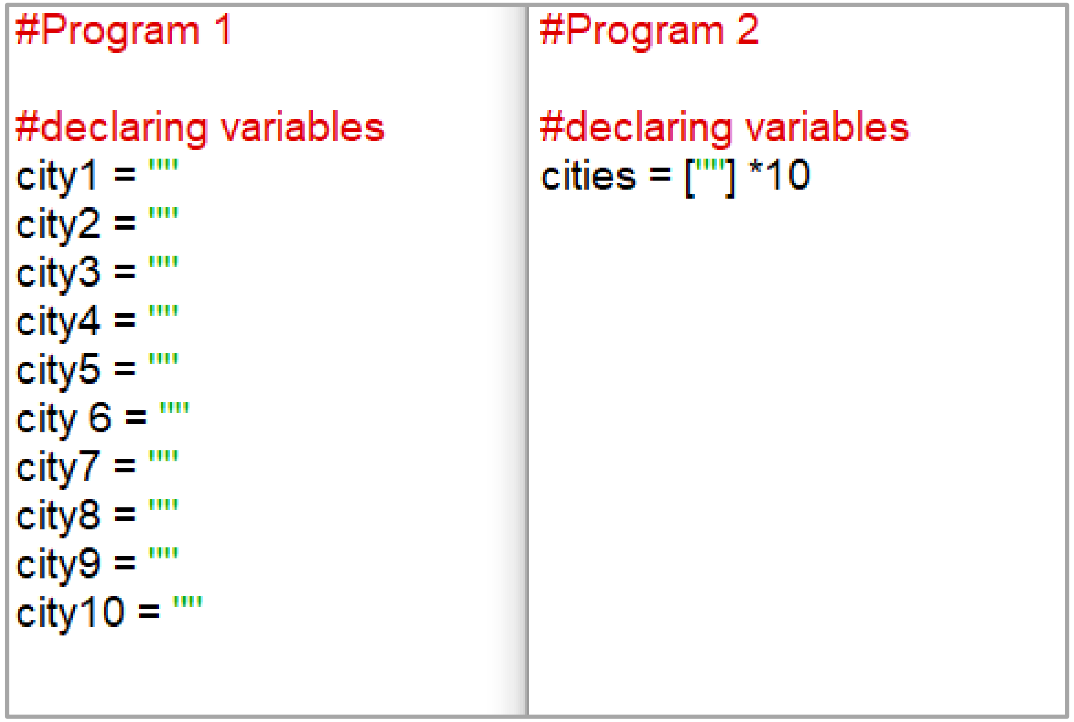
```
#Program 2

if age >= 65:
    ticketCategory = "concession"
elif: age >= 18 and age < 65:
    ticketCategory = "adult"
else:
    ticketCategory = "child"
```

# Efficiency: Arrays

Comparing these two programs, we can see that Program 2 is more efficient.

Program 1 requires many more lines of code to be written, which means more processing required during translation. It also requires more RAM to store the name and value of each piece of data.



```
#Program 1

#declaring variables
city1 = ""
city2 = ""
city3 = ""
city4 = ""
city5 = ""
city 6 = ""
city7 = ""
city8 = ""
city9 = ""
city10 = ""
```

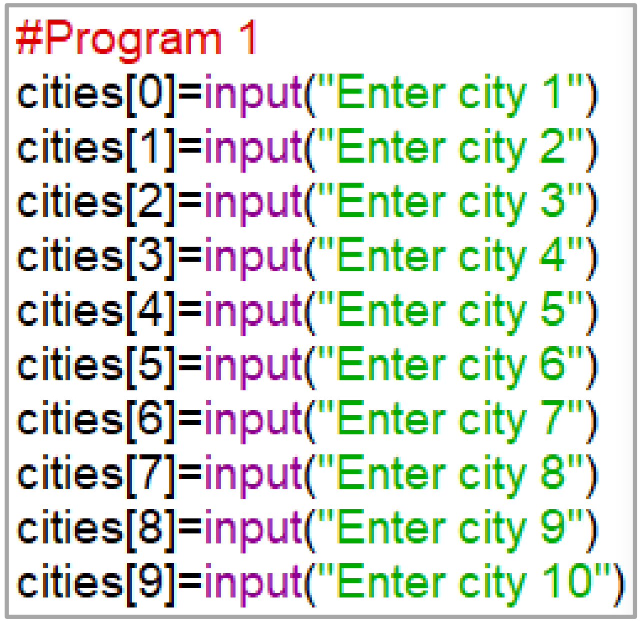
```
#Program 2

#declaring variables
cities = [""] *10
```

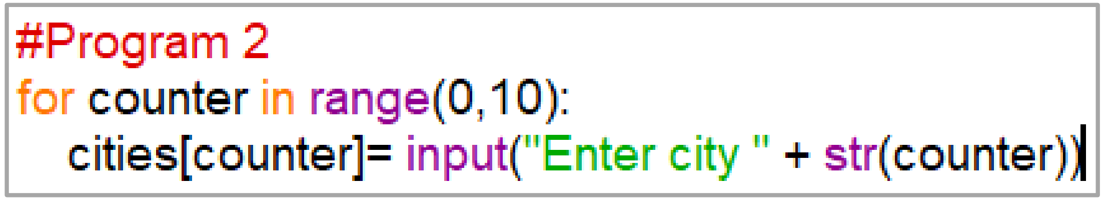
# Efficiency: Fixed Loops

Comparing these two programs, we can see that Program 2 is more efficient.

Program 1 requires many more lines of code to be written, which means more processing required during translation.



```
#Program 1
cities[0]=input("Enter city 1")
cities[1]=input("Enter city 2")
cities[2]=input("Enter city 3")
cities[3]=input("Enter city 4")
cities[4]=input("Enter city 5")
cities[5]=input("Enter city 6")
cities[6]=input("Enter city 7")
cities[7]=input("Enter city 8")
cities[8]=input("Enter city 9")
cities[9]=input("Enter city 10")
```



```
#Program 2
for counter in range(0,10):
    cities[counter]= input("Enter city " + str(counter))
```

**Note**: conditional loops should not be discussed in terms of efficiency as they do not replace an inefficient construct.

# Evaluation: Robustness

Robustness refers to how well your program is able to cope with unexpected input. Unexpected data can be provided by files or directly from users.

Programs use input validation to ensure that users provide acceptable data, making them robust.

You can ensure that your program is robust by using exceptional test data during the Testing phase of development.

# Evaluation: Readability

It’s important that your code is readable to help you and others understand what you intend your code to do.

Most of the time, code bases are written by dozens of people so it’s important that your code is readable so it’s easy for *any* member of the team to make changes and fix bugs.

# Evaluation: Readability

Readable code should:

- Use meaningful variable names
- Use internal commentary to help explain complex parts of the code
- Have white space between different sections of code to separate different pieces of logic
- Use indentation to help to identify loops and selection statements