13. TESTING

All computer programs will need to be tested to ensure that they work as expected, i.e. produce the expected outputs when a range of inputs are entered, and secondly to ensure that, as much as possible, all possible errors have been found and corrected or managed through the use of robust coding techniques.

There are two ways in which a program can be tested:

1. Iterative testing
2. Final or terminal testing

13.1 ITERATIVE TESTING

Iterative testing is carried out as an integral part of developing a program; we call it iterative as it is a process that involves the continuous testing and improvement of each individual function or module to ensure that each part performs as expected before moving on to the next.

For example, running this code and discovering that it results in an infinite loop, changing the indentation of the incrementing variable is an example of iterative testing.

```
def numberLoop():
    """ while loop example"""
    number = 1           # initial value of the variable
    while number <= 10: # the condition to exit the loop
        print(number)
    number +=1           # incrementing the value of the variable


numberLoop()
```

```
def numberLoop():
    """ while loop example"""
    number = 1           # initial value of the variable
    while number <= 10: # the condition to exit the loop
        print(number)
        number +=1       # incrementing the value of the variable


numberLoop()
```


13.2 SYNTAX, RUNTIME AND LOGICAL ERRORS

There are three types of error that you may come across when programming:

<table>
<tr><th>Error Type</th><th>Explanation and examples</th></tr>
<tr><td rowspan="2">Syntax</td><td>This means that your code does not follow the rules of the programming language. You may have missed a bracket, forgotten to add speech marks, or have a spelling error in a key word.</td></tr>
<tr><td>

```
>>> Print("Hello World!")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    Print("Hello World!")
NameError: name 'Print' is not defined
```

</td></tr>
<tr><td rowspan="2">Runtime</td><td>This means that when the program runs an error occurs. Examples could be trying to divide a number by 0 or trying to open a file that does not exist.</td></tr>
<tr><td>

```
>>>
Traceback (most recent call last):
  File "C:\Users\ExampleFiles\readFileExample3.py", line 9, in <module>
    readFile3()
  File "C:\Users\ExampleFiles\readFileExample3.py", line 5, in readFile3
    with open('shopping.txt','r')as myFile:
FileNotFoundError: [Errno 2] No such file or directory: 'shopping.txt'
```

</td></tr>
<tr><td rowspan="2">Logical</td><td>This means that your program will run but will give unexpected results. Examples are using mathematical operators incorrectly, e.g. > instead of <, or forgetting to put brackets around a calculation (BIDMAS/BODMAS).</td></tr>
<tr><td>

```
>>> x = 3
>>> y = 4
>>> average = x + y / 2
>>> print(average)
5.0
```

The answer should be 3.5; we need to ensure that the calculation happens in the right order to correct this logical error.

```
>>> average = (x + y) / 2
>>> print(average)
3.5
```

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

13.3 REDUCING SYNTAX ERRORS

You will make a lot of syntax errors when you are learning how to program. They are easy to spot as the interpreter will highlight where the error is. In this example, the interpreter indicates that the error is EITHER at the start of the line OR at the end of the previous line of code.

```
>>> print("I want to print this"
      but I have not closed the brackets

SyntaxError: invalid syntax
```

Some simple rules to follow:

- If you open a bracket, make sure you close it.

```
>>> name = "John"
>>> print("Hello {0}".format(name))
Hello John
>>>
```

- If you use speech marks, make sure you close them.
- Use the correct operator, e.g. = means assign a value, == means equality.
- Make sure your code is indented correctly, e.g. in control structures.

```
x = 5
y = 8

if x < y:
    print("X is bigger")
    else:
        print("Y is bigger")
```

```
x = 5
y = 8

if x < y:
    print("X is bigger")
else:
    print("Y is bigger")
```

13.4 REDUCING RUNTIME ERRORS

These are harder to spot and, therefore, avoid, as your code will pass the syntax test and Python will understand everything you have written but give you the wrong answers or no answers at all!

This code should read in our file and print it – *can you spot the error?*

```
def read_file1():
    """reads in file line by line"""
    with open('shopping.txt', 'r')as myFile:
        line = myFile.readline()
        while len(line) != 0:  # while there is data in the line
            print(line, end = '')
            line = myFile.readline()# read the next line

read_file1
```

Some simple rules to follow:

- Test each part of your program as you write it; it will be easier to spot your errors.
- Use a print statement to check what is happening to a variable as your code is executed.
- Check your code carefully.

13.5 REDUCING LOGICAL ERRORS

These can be even harder to spot than runtime errors as the interpreter will not give you any error messages or warnings. If you have calculations in your code, work them out by hand so you know what results you should get from the program.

- Use print statements to see how the variable values change at different points in your code.
- Test each part of your program separately to pinpoint where the error could be.
- Manually trace the execution of your program using a trace table.

TRACE TABLES

A trace table, sometimes called a 'dry run', is a manual method of testing an algorithm to ensure there are no logic errors. If we look at the example above, the LOGIC error would be spotted easily using this manual method.

```
1  def numberLoop():
2      """ while loop example"""
3      number = 1       # initial value of the variable
4      while number <= 10: # the condition to exit the loop
5          print(number)
6      number +=1   # incrementing the value of the variable
7
8
9  numberLoop()
```

Line	number	number <= 10	number +=1	OUTPUT
3	1			
4		True		
5				1
4		True		
5				1
4		True		
5				1

The trace table clearly shows that the value of the variable 'number' is never incremented inside the while loop, making this an infinite loop.

Changing the logic of the program to ensure that the variable 'number' is incremented INSIDE the loop can again be proved using a trace table.

```
1  def numberLoop():
2      """ while loop example"""
3      number = 1       # initial value of the variable
4      while number <= 10: # the condition to exit the loop
5          print(number)
6          number +=1   # incrementing the value of the variable
7
8
9  numberLoop()
```

Line	number	number <= 10	number +=1	OUTPUT
3	1			
4		True		
5				1
6			2	
4	2	True		
5				2
6			3	
4		True		
5				3
6			4	

This is an easy way of checking the logic of your code before you actually write the program code.

13.6 HANDLING ERRORS THROUGH ROBUST CODE

Some of the errors shown in the Dealing with Errors section have specific names, e.g. NameError, FileNotFoundError. We can write robust code to deal with these named errors; this is called 'exception handling'. This means that we can ensure that our program does not just crash, but prints a meaningful error message, and we can control what happens next.

13.7 VALUE ERROR

In this example, the code is checking that input value is an integer; this is called a 'try statement'.

```
def get_input():
    """ get integer input"""
    while True:
        try:
            num = int(input("Please enter a whole number: "))
            print("The number you entered was {0}".format(num))
            break
        except ValueError:
            print("That was not an integer, please try again.")

    return num

num = get_input()
```

```
Please enter a whole number: f
That was not an integer, please try again.
Please enter a whole number: 3.25
That was not an integer, please try again.
Please enter a whole number: 6
The number you entered was 6
>>>
```

13.8 ZERO DIVISION ERROR

```
def divide_nums():
    """ divide x by y and print result"""
    try:
        x = int(input("Please enter a number to divide: "))
        y = int(input("Please enter a number to divide by: "))
        print("The result of {0}/{1} is {2}".format(x, y, x / y))
    except ZeroDivisionError:
        print("Cannot divide by 0!")
        y = 1
```

```
>>>
Please enter a number to divide: 15
Please enter a number to divide by: 3
The result of 15/3 is 5.0
```

If we try to enter a 0, the interpreter will 'catch' the exception and handle it by printing the error message and changing the value of the variable y to 1.

```
>>>
Please enter a number to divide: 15
Please enter a number to divide by: 0
Cannot divide by 0!
```


13.9 IO ERROR

IO errors occur when we try to create a file object that does not exist or to write the file to a storage area which is full. At GCSE level you should be able to write code to deal with the first error, i.e. the file is not there OR the filename supplied is incorrect.

In this example, the name of the file is incorrect, the exception is handled and the error message is displayed.

```
def read_file():
    """reads in file line by line"""
    try:
        with open('shoppin.txt','r')as myFile:
            line = myFile.readline()
            while len(line)!= 0:  # while there is data in the line
                print(line, end = '')
                line = myFile.readline()# read the next line
    except:
        print("File does not exist...")

read_file()
```

```
>>>
File does not exist...
>>>
```

13.10 MULTIPLE EXCEPTIONS

We can also write code to handle multiple exceptions like the following example. This example covers the KeyboardInterruptError, which happens when a user hits the Delete key or CTRL+C and combines this with the ValueError seen previously.

The example shows how you might use this in a menu for a program and makes use of the sys.exit() built-in function to end the program.

```
import sys


def menu():
    ''' gives the user menu options'''
    print('\n\n')
    print("*" * 35)
    print("\t Python Game")
    print("*" * 35)
    options = [1, 2, 3, 4]

    print("""Please choose from the following menu options:
    \n\t1)Enter name\n\t2) Read rules\n\t3) Play\n\t4) Exit\n""")
    menu_choice = 0
    while menu_choice not in options:
        try:
            menu_choice = int(input("Please enter 1,2,3 or 4: \n"))
            break
            print("That is not a valid menu choice")
        except (ValueError,KeyboardInterrupt):
            print("You must enter 1,2,3 or 4- you entered an invalid character\n")

    return menu_choice
```



```
def main():
    ''' runs all functions'''
    menu_choice = 0
    while menu_choice != 4:
        menu_choice = menu()
        if menu_choice == 1:
            print("You have chosen to enter your name")
        elif menu_choice == 2:
            print("You have chosen to read the game rules")
        elif menu_choice == 3:
            print("You have chosen to play")
        else:
            print("You have chosen to quit")
            sys.exit()

main()
```

EXERCISE 31: ERROR HANDLING

1. What type of error is shown in this code snippet?

```
number = 1
while number < 13:
    print("{0} squared = {1}".format(number,number * number))
```

2. Correct the code so that it will print the squared numbers from 1 to 12.

3. Rewrite the code below so that there are no exception errors when trying to open this non-existent file.

```
def writeFile():
    """write data to a file"""
    myFile = open('new_file.txt','r')
    myFile.write("Little Bo Peep has lost her sheep ")
    myFile.write("\n")
    myFile.write("And doesn't know where to find them.")
    myFile.write("\n")
    myFile.close()
```

13.11 TEST PLANS

Final, or terminal, testing happens when you have completed your whole program, having worked through iterative testing of each component part and written robust code and error handling routines to deal with any potential errors.

Test plans usually follow a common table format like this:

Test Number	Test Description	Data Input	Expected Outcome	Actual Outcome	Improvements

It is important to test all your code as you complete each function or section of your program. Some of your tests may not require any data input; for example, ensuring that a menu for a game displays the correct options for the game.

There are three categories of test data that you need to use in your test plan:

- **Normal data** – data that is in the normal range
- **Extreme/Boundary data** – data that is at the edge of the acceptable range
- **Erroneous/Invalid data** – data that is the wrong value or the wrong data type

Each time you create a new test you need to do the following:

- Give a clear description of what is being tested
- Specify what the data input is and the type of data being entered
- Give a clear description of what you expect will happen when the test is performed
- Run the test and take a print-screen of the results
- Give a clear description of what actually happened and reference the print-screen evidence
- If the test did not work as expected, you need to explain what improvements are needed AND run the test again to prove that the code now works as expected

The following pages have some example test plans and testing evidence for the Magic Square Game:

TESTING THE MAGIC SQUARE GAME:

Test Number	Test Description	Data Input	Expected Outcome	Actual Outcome	Improvements
1	Test that the game board displays correctly when the game opens.	N/A	That the game board will display an 8 × 8 grid with numbers across the top of each column and letters at the start of each row.	The board displays correctly as expected. See results of test 1.	The instructions for the player should be below the game board so that they are clearer.

Results of Test 1:

```
  1  2  3  4  5  6  7  8
A 0  0  0  0  0  0  0  0
B 0  0  0  0  0  0  0  0
C 0  0  0  0  0  0  0  0
D 0  0  0  0  0  0  0  0
E 0  0  0  0  0  0  0  0
F 0  0  0  0  0  0  0  0
G 0  0  0  0  0  0  0  0
H X  0  0  0  0  0  0  0 You can move your player up, down or stay on the same row
.Please enter U,D or S:
```


Test Number	Test Description	Data Input	Expected Outcome	Actual Outcome	Improvements
2	Test that the game board displays correctly when the game opens.	N/A	That the game board will display an 8 × 8 grid with numbers across the top of each column and letters at the start of each row. The player instructions display below the grid.	The board displays correctly as expected. See results of test 2.	None

Results of Test 2:

```
  1  2  3  4  5  6  7  8
A 0  0  0  0  0  0  0  0
B 0  0  0  0  0  0  0  0
C 0  0  0  0  0  0  0  0
D 0  0  0  0  0  0  0  0
E 0  0  0  0  0  0  0  0
F 0  0  0  0  0  0  0  0
G 0  0  0  0  0  0  0  0
H X  0  0  0  0  0  0  0

You can move your player up, down or stay on the same row.
Please enter U,D or S:
```

Test Number	Test Description	Data Input	Expected Outcome	Actual Outcome	Improvements
3	Test the data input for player vertical movement.	Erroneous Data 'x'	The program will print an error message and ask for input again.	The results of the test are as expected. See results of test 3.	None

Results of Test 3:

```
You can move your player up, down or stay on the same row.
Please enter U,D or S: x
That is not a valid option

You can move your player up, down or stay on the same row.
Please enter U,D or S:
```


Test Number	Test Description	Data Input	Expected Outcome	Actual Outcome	Improvements
4	Test the data input for player vertical movement direction.	Normal Data 'U'	The program will accept the input and ask how many squares to move.	The results of the test are as expected. See results of test 4.	None

Results of Test 4

```
You can move your player up, down or stay on the same row.
Please enter U,D or S: U
Please enter the number of squares to move:
```

Test Number	Test Description	Data Input	Expected Outcome	Actual Outcome	Improvements
5	Test the data input numbers of squares to move.	Extreme Data 7	The program will accept the input and ask whether there is any horizontal movement.	The results of the test are as expected. See results of test 5.	None

Results of Test 5

```
Please enter the number of squares to move: 7
You can move your player left, right or stay on the same row.
Please enter L,R or S: s
```

