12. DEFENSIVE DESIGN

What is defensive design? It means thinking about all the possible ways in which an end user may interact with a program so that no matter what data is entered or what actions the end user takes the program will continue to run as expected.

There are three main areas which are crucial in defensive design:

- Ensuring that the program can cope with any unexpected data inputs or actions; for example, entering a decimal value where an integer is expected.
- Ensuring the code is written in a style that is easy to understand and read so that any improvements are easy to make at a later date.
- Ensuring that any bugs or potential errors in the program code have been eliminated through detailed testing with a wide range of data.

Defensive design incorporates:

- Validation
- Sanitisation
- Authentication
- Maintenance
- Testing

12.1 VALIDATION

When you are writing your own programs, it is important to ensure that the data entered by the end user is reasonable or sensible to ensure that your program does not crash or produce unexpected results. This technique is called 'validation'.

Example: You may have registered with a website where you have been asked for your date of birth. This is commonly achieved with drop-down lists which restrict your data entry to the number of days in a month and months in the year, and a range of years for people of all ages – it will not allow you to be 150 years old as that is not reasonable or sensible!

Validation does not check whether the data entered is correct, only that the values are reasonable and within any boundaries you have set, e.g. an email address must include the @ symbol.

What should be checked?

- **Type check:** Data type entered is the correct type, e.g. a person's age is an integer and a person's name is a string
- **Range check:** Data entered is within the correct range, e.g. a person's age is not a negative number
- **Length check:** Data entered is the correct length, e.g. telephone numbers
- **Format check:** Data entered is in the right format, e.g. upper or lower case, to ensure that equality checks work as expected
- **Presence check:** Data has actually been entered; for example, a username.

There are a number of techniques we can use in Python to ensure data entered is reasonable or sensible. We can use programming techniques such as while loops, forcing the entry to be upper or lower case or checking a value is within an acceptable range.

We can also use these additional built-in functions for strings:

Function	Explanation
isalpha()	Returns true if all characters in a string are letters
isdigit()	Returns true if all characters in a string are numbers
isalnum()	Returns true if all characters are letters (a–z) in upper/lower case or numbers (0–9)
isdecimal()	Returns true if characters in a string are decimals
isnumeric()	Returns true if characters in a string are numbers
isupper()	Returns true if characters in a string are upper case
islower()	Returns true if characters in a string are lower case

It would seem that there are three built-in functions that do the same thing; however, technically they are different, but the reasons why is beyond the scope of this resource and relates to Unicode characters.

This example shows how the three of them work for a string with a decimal point or a space.

```
>>> s = "12.34"
>>> print(s.isdigit())
False
>>> print(s.isdecimal())
False
>>> print(s.isnumeric())
False
>>> s = "12 34"
>>> print(s.isdigit())
False
>>> s = "1234"
>>> print(s.isdigit())
True
>>> print(s.isdecimal())
True
>>> print(s.isnumeric())
True
```


Example: type check, length check and presence check

In the example below, the code checks that the data input from the user is letters only and that the string is at least two characters in length.

```
1   def get_name():
2       """asks for user's name"""
3       name = ''
4       invalid_name = True                    # variable to control the while loop
5       while invalid_name:
6           name = input("Enter name :")       # ask for input inside the loop
7           if name == "":
8               print("The data entry is blank,please enter your name")
9           elif not name.isalpha():
10              print("Your name can only contain letters.")
11          elif len(name) < 1:
12              print("Your name must have more than 1 letter")
13          else:
14              print("Valid name entry")
15              invalid_name = False
16      return name
17
18
19  def main():
20      """runs all functions"""
21      name = get_name()
22      print("Hello {0}".format(name))
23
24
25  main()
```

(Code shown in PyCharm IDE to display line numbers)

In this example, the while loop on line 5 controls the three checks for data input; the loop will continue until the string entered has passed the three validation checks, i.e. that the data entered is not an empty string, that the string for the user name consists of letters only, and that it is more than one letter.

Line 6 uses the built-in function isalpha() with the logical operator NOT; this will evaluate to True if the string contains both letters and numbers.

When these three checks have been passed in the else part of the selection statement, the Boolean value of the variable controlling the loop is changed to False and the loop ends.

Notice on line 5 that we do not need to use any equality operator in our while statement when we are using Boolean values.

The isdigit() built-in function ONLY applies to strings; if you wanted an integer or a float input you would cast the input to an integer or float:

```
>>> hw_score = int(input("Enter score: "))
Enter score: 55
>>> type(hw_score)
<class 'int'>
>>> shoe_size = float(input("Enter shoe size: "))
Enter shoe size: 5.5
>>> type(shoe_size)
<class 'float'>
>>>
```


Example: range check and type check

How could we use isdigit() in our code? Look at this example:

```
1   def get_mobile_no():
2       """gets mob phone number as pseudonumber"""
3       tel_number = ''
4       invalid_no = True
5       while invalid_no:
6           tel_number = input("Enter phone number: ")
7           if len(tel_number) in range(9,12) and tel_number.isdigit():
8               print("Valid entry")
9               invalid_no = False
10          else:
11              print("You must enter a number must be between 9 and 11 digits")
12      return tel_number
13
14
15  def main():
16      """runs all functions"""
17      tel_number = get_mobile_no()
18      print("You entered {0}".format(tel_number))
19
20
21  main()
```

When we want to store telephone numbers or mobile phone numbers, these commonly start with a 0; remember pseudo-numbers?

If we used a number data type then the leading 0 will be lost. As you can see on line 7, we have combined our checks so that the number entered must be between 9 and 11 characters in length AND all numbers.

EXERCISE 29: VALIDATION

1. The code shown on the right should only accept data in the range 0–100.

   Amend the code so that the user cannot continue until a value in the correct range has been entered.

   Your code must display an error message if a value outside this range is entered.

```
def get_score():
    """asks for test score"""
    test_score = int(input('Enter test score: '))

    return test_score

def main():
    """runs all functions"""
    test_score = get_score()
    print("These are the GCSE grade boundaries:")
    if test_score >= 90:
            print('Your grade is 9')
    elif test_score >= 80:
            print('Your grade is 8')
    elif test_score >= 70:
            print('Your grade is 7')
    elif test_score >= 60:
            print('Your grade is 6')
    elif test_score >= 50:
            print('Your grade is 5')
    elif test_score >= 40:
            print('Your grade is 4')
    elif test_score >= 30:
            print('Your grade is 3')
    elif test_score >= 20:
            print('Your grade is 2')
    else:
            print('Your grade is 1')

main()
```


12.2 SANITISATION

An important part of defensive design is ensuring that data is in the correct format before the program moves on to the next step.

There are two examples of input sanitisation shown in the example here:

```
def display_menu():
    """displays the menu and asks for play choice"""
    # use print to display the menu

    print("\t\tGame Menu\n")
    print("\t\tE - Enter Name\n\t\tP - Play Game\n\t\tQ - Quit")

    valid_option = ['E', 'P', 'Q']

    while True:
        selection = input("Please enter your choice: ").upper()
        selection = selection[0]
        if selection in valid_option:
            break
        else:
            print("That is not a valid choice")
    return selection
```

Example 1:

The input string is forced to upper case so that any lower-case data entry will match the options specified in the list.

Example 2:

The first character of the input string is selected for checking, in case the end user enters ‘play’ or ‘quit’ rather than a single character.

Input sanitisation could also include ‘masking’ data entry, such as when you enter a password to login to a computer. There are techniques in Python 3 to do this, but they are beyond the scope of this GCSE resource.

12.3 AUTHENTICATION TECHNIQUES

As explained previously, validation techniques only check that the data entered is sensible or reasonable, not that it is correct or valid. Authentication involves having checks to ensure that the person trying to access a network or system is who they say they are; this is usually achieved through the use of usernames and passwords. You will experience authentication procedures each time you logon at school or use remote access from home. The network manager will maintain an encrypted file of all usernames and their respective matching passwords, together with the level of access to the system, i.e. the areas of the network you are allowed to see.

At GCSE level you are not expected to use any encryption to protect usernames and passwords, and we will, therefore, store the data as plain text.

This example on page 7 uses a 2D array which is hard-coded into the program, and nested loops to examine each username or password.

Lines 3–4 Creates the 2D array; each user has a username, password and security level.

Lines 7–9 Variables are initialised to be used in the condition-controlled loop.

Line 11 The while loop has two conditions, giving the user three attempts to enter a valid username.

Lines 12–13 The count is incremented on Line 12 and the input requested in Line 13.

Lines 14–26 The nested FOR loop loops through the main array to look at each of the sub-arrays, Line 16 then looks at each item in each sub-array. The nested IF statements check whether the username entered matches the data at position [0] in the sub-arrays.

If it does match, then the variable found is changed to True, meaning one of the conditions controlling the WHILE loop has now changed. The user is prompted for the password (Line 20). Line 21 uses the password which matches the username (user Position [1]) to check that the password entered is a match.

Lines 30–31 If the user has incorrectly entered a username three times, the error message is printed.

EXERCISE 30: AUTHENTICATION

Part A: The authentication code example using 2D arrays checks that the username is entered correctly after three attempts; however, it only gives the user one attempt at the password.

Improve the code, using functions and parameter passing, so that each part of the process allows the user three attempts before displaying the message on Line 31.

Part B: The current authentication code does not allow for new users to be added or any former users to be removed, which is clearly a security issue.

Improve your solution to incorporate a method to ensure that the list of system users can be updated without needing to edit the original program code.


```
# Authentication routines using 2D arrays

system_users = [["Username", "Password", "Level"], ["13Garcia", "qBkEKg99", "Admin"], ["15Connor", "R5mF92Sq",
                "Standard User"], ["18Khan", "4xkY5Md2", "Standard User"], ["16Simpson", "8B73gQB7", "Standard User"]]

# set variables to control the loop
found = False
valid = False
count = 0

while not found and count < 3:
    count += 1
    u_name = input("Enter your username: ")
    # loop through 2D array to check if the user exists
    for user in system_users:
        for each in user:
            if each == u_name:
                found = True
                print("Username valid")
                u_pwd = input("Enter password: ")
                if user[1] == u_pwd:
                    valid = True
                    print("Password accepted: " + str(user[2] + " Level"))
                    break
                else:
                    print("Invalid username or password")


# If 3 attempts to enter valid username are not successful
if not found:
    print("Three attempts-Invalid username or password")
```

(Code shown in PyCharm IDE to display line numbers)

12.4 MAINTENANCE

Making your programs easy to maintain is a key feature of defensive design, which should be considered and planned at the outset. This is important because, in the future, the program may need additional features or need to be adapted to work with a new database, etc.

For example, many government systems and banking systems still use a language called COBOL (Common Business-Oriented Language). There are fewer and fewer programmers who understand it; therefore, it is vital that the code has been clearly documented and explained.

The best way to ensure that your program is easily maintainable is to use:

- Comments
- Suitable variable names
- Clear indentation in the code

COMMENTS

There are several ways in which comments can be used to provide information about your program.

- Using docstrings in functions to explain what they do
- Explaining any complex sections of code
- Providing breaks between different areas of long, complex sections of code

```
def get_name():
    """validates username entry, checks for blank entry
    numbers in the string or single character entry """

    name = ''
    invalid_name = True                  # variable to control the while loop
    while invalid_name:
        name = input("Enter name :")     # ask for input inside the loop
        if name == "":
            print("The data entry is blank,please enter your name")
        elif not name.isalpha():
            print("Your name can only contain letters.")
        elif len(name) <= 1:
            print("Your name must have more than 1 letter")
        else:
            print("Valid name entry")
            invalid_name = False
    return name


def main():
    """runs all functions"""
    name = get_name()
    print("Hello {0}".format(name))


main()
```


In this more complex example, the code has been divided into separate tasks with more detailed comments:

```
# ------------------------------------------------------------------------------------------------------------
# Task1- read in the file


def read_file():
    """reads in the text file and splits it into two lists. Then choose the word to be guessed,
    the index of the word and the definition from the chosen word index."""

    # create the empty lists for the keyword and definitions

    keyword = []
    defined = []
    temp = []

    # use error trapping when reading into the two lists

    try:
        my_file = open("keywords.txt", 'r')
        # read into a temp list

        try:
            for eachLine in my_file:
                temp.append(eachLine.strip()) # strips the /n newline code

                # list comprehension makes 2 new lists by copying odd and even lines to two new lists
                keyword = [temp[i] for i in range(len(temp)) if i % 2 == 0]
                defined = [temp[i] for i in range(len(temp)) if i % 2 == 1]

        finally:
            my_file.close()
    except IOError:
        print("File does not exist...")

    # create a list the same length as keyword to count when guessed twice
    matched = []
    # create guessed empty list- this will be used to count which questions have been answered twice
    guessed = []
    # create a completed list to check against when looping through questions
    completed = []

    # return multiple objects in a list
    return defined, keyword,matched, guessed,completed

# ------------------------------------------------------------------------------------------------------------
# Task2- find a random keyword
```

SUITABLE VARIABLE NAMES

As demonstrated in the example above, it is important to use variable names that will make sense if you need to return to the program at a later date. Additional comments have been used to explain exactly what each variable will be used for in the program.

The use of appropriate variable names in Python is also covered in ***Chapter 2 – Variables & Assignment***.

INDENTATION

Python indents the code you write automatically, providing you follow the correct syntax. Sometimes the meaning of your code can be altered; for example, in nested loops if you incorrectly indent code such as incrementing a count variable.

Example of incorrect indentation:

```
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()
```

What will be the result of running this code?

The correct indentation will avoid an infinite loop!

```
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()
```

