OCR Computer Science GCSE

2.3 – Producing robust programs

Advanced Notes

2.3.1 Defensive design

What is Defensive Design?

Defensive design refers to designing and creating programs which are able to handle unexpected or erroneous data and/or inputs by anticipating misuse. It is highly likely that users will make errors or bad inputs when using a program.

Defensive design is important because it ensures:

- The number of bugs in a program are reduced
- The program behaves as expected, regardless of user input
- Any possible errors are accounted for

Authentication is the process of determining the identity of a user, usually through a username and password. The authentication process verifies that a username exists, and the password entered is correct and linked to the username.

Input Validation

Validation is the process of checking that data is appropriate for its use, so that it can be correctly processed without any errors.

Validation	Description	Examples (Python)

Presence check	Ensures that data has been entered and not blank.	`name = input("Enter your name")`
		`if name == "":`
		`  print("Invalid")`

Range check	Ensures that an input falls within the required range.	`num = int(input("Enter a number less than 10"))`
		`if num >= 10:`
		`  print("Too large!")`

Length check	Ensures a specified number of characters have been entered.	`password = input("Enter a password, minimum 8 characters.")`
		`if len(password) < 8:`
		`  print("Password too short!")`

Maintaining Programs

It is incredibly important to maintain programs to ensure that they function as intended, are secure, and can be improved over time.

There are many ways of maintaining programs, such as naming conventions and adding comments.

- **Use of sub-programs:**
  Sub-programs (functions and procedures) help to break down a program into manageable chunks. By doing so, code is not repeated as the sub-programs can be reused. Additionally, they can be easily tested and improved without having to rewrite or adjust entire sections of code throughout the main-program.

- **Naming conventions:**
  All variables, functions and procedures should be appropriately named to signify their purpose. This makes it easier to track which variable is which, as well as what its purpose is in the program. ← Poor naming

  - This example has poor variable names, it can be hard to know which is which

```
num1 = float(input("Enter the mass"))
num2 = float(input("Enter the volume"))
num3 = num1 / num2
print("The density is ", num3)
```

  - This example has clear variable names, it is easy to know what each one is

```
mass = float(input("Enter the mass"))
volume = float(input("Enter the volume"))
density = mass / volume
print("The density is ", density)
```

- **Indentation:**
  Indentation is not only required for the syntax of most programming languages, but also makes it easier to visualise and understand what sections code belongs to.

- **Commenting:**
  Using comments is an easy way to describe what each line of code, or what each function is responsible for, making it easy to come back later on and quickly understand what the code does. Comments in OCR Pseudocode are written using '//'. The hash key, '#', is used in Python.

  - Example (pseudocode):

```
function addNum(num1, num2)
  sum = num1 + num2 // Adds two numbers
  print(sum) // Prints the sum
endfunction
```

2.3.2 Testing

The Purpose of Testing

All programs should be tested to ensure that they are robust, secure, and work as intended.

Testing is usually **destructive**, meaning that you should aim to find as many errors through rigorous testing, rather than just showing that a program works, which it may in specific cases, but may also contain bugs.

Testing takes place at two stages of software development:

- **Iterative Testing:**
  This type of testing happens throughout development, usually testing individual sub-programs as they are created, and then using the results from testing to make any changes or improvements to ensure they work as intended and without bugs.

- **Final Testing:**
  Testing which takes place at the end of development, which aims to test the functionality of the entire program and check for any bugs.

    - Alpha testing is carried out by developers in-house to fix any remaining issues.

    - Beta testing is performed by groups of real end-users, commonly seen in large software releases such as in games.

Syntax and Logic Errors

Syntax errors are errors which break the grammatical rules of the programming language, which stop the program from running.

Common syntax errors include:

- Missing brackets, quotation marks or colons
- Misspelling keywords, e.g `prnt` instead of `print`
- Using variables which have not been declared
- Incorrect indentation

Syntax errors are easy to fix as they are often identified and flagged up by an IDE or code editor since the code won't run.

Logic errors are errors in the program's design or logic, which cause it to produce an unexpected or incorrect output, even if the program runs or the syntax is correct.

For this reason, logic errors are often a lot harder to spot as they rely on you, as the programmer, to identify where the program is not working as intended and is also why testing is so important.

- Example (Python):

```
distance = float(input("Enter the distance"))
time = float(input("Enter the time"))
speed = time / distance
```

  There is actually a logic error above! Speed should be calculated as:
  speed = distance / time

Test Data

Types of Test Data:

Type	Purpose	Example (Range: 1–10)
Normal	Typical input	`5`
Boundary	On the edge of valid range	`1 and 10`
Erroneous	Invalid input	`-1`, `eleven`, `"abc"`

Normal and boundary test data should be accepted by the program, whereas erroneous data should be rejected without causing an error.

Selecting Suitable Test Data

You should be able to:

- Identify appropriate test data for a given input field
- Justify why it's used (e.g. "Boundary data ensures edge cases work")

Refining Algorithms

After an algorithm has been developed and tested, it may be refined over time to improve its functionality, or its capacity, as it may become increasingly popular. Changes may include:

- Fixing problems found throughout testing
- Adding additional functionality, or improving existing functionality
- Making the program more efficient by removing unnecessary components