

ACCA4020 CW Moyo

by Rolinhlanhla Moyo

Submission date: 08-Jun-2026 05:59PM (UTC+0100)

Submission ID: 285509864

File name: 67850_Rolinhlanhla_Moyo_ACCA4020_CW_Moyo_727104_1888469628.docx (1.04M)

Word count: 3261

Character count: 17308

DIGITAL INNOVATION IN PARKING-LOT MANAGEMENT

Module: ACCA4020

Module Name: Introduction to Programming

Student Name: Rolinhlanhla Sebastian Moyo

Student ID: 2522318

Course: Foundation (Computing)

University: University Of Wales Trinity Saint David (UWTSD)

Submission Date: May 5th 2026

TABLE OF CONTENTS

1.1	The program	3
1.2	Program tests, Results, and observations.....	Error! Bookmark not defined.
1.3	Observation analysis.....	11
2.1	Implementation of correct code.....	12
2.2	Re testing and verification	13
3.1	Difficulties faced without advanced data structures.....	17
4.1	Most challenging programming concepts	18
4.2	How were the challenges overcome.....	18
4.3	Application in the task	19
4.4	Most useful test case	19



1. INTRODUCTION

Digital innovation is becoming increasingly important in modern university environments, allowing facilities to operate more efficiently, sustainably, and effectively. As part of a university-wide initiative to modernise campus infrastructure and improve services, the development of a prototype Smart Parking Lot Management System has been requested. This system is intended to be deployed within the university's multi-zone car park, providing real-time monitoring of occupancy levels, generating utilisation statistics, and alerting management when zones approach maximum capacity. The aim is to reduce congestion, optimise space usage, and enhance the overall experience for staff, students, and visitors.

The purpose of this report is to document the full development lifecycle of the system as required by the assessment brief. The work begins by implementing the logic represented in the provided flowchart exactly as it was designed by another team, even where potential issues were suspected. Through systematic testing using defined test cases, logical flaws, operational errors, and usability problems are identified and analysed in detail. Once these issues are clearly understood, corrections are applied to the source code to ensure the system operates reliably, accurately, and in line with the stated requirements.

Beyond simple correction, the design is evaluated to determine how efficiency, maintainability, and scalability can be improved through the application of more advanced programming structures and techniques. The limitations of using only basic variables and control structures are explored, and alternative approaches are proposed that would make the system suitable for real-world use.

Finally, a reflection is provided on the learning process throughout the module, examining which programming concepts were found most challenging, how understanding was developed, and how practical testing changed the perception of system design. The report concludes with an assessment of how the skills acquired have been applied to complete this project successfully.

The work is presented in four distinct tasks, following the specifications and marking criteria provided, ensuring all learning outcomes are addressed thoroughly.



2. IMPLEMENTATION OF ORIGINAL FLOWCHART LOGIC

2.1 THE PROGRAM

Based **solely** on the logic modelled in the provided flowchart, the following code was developed. It is the initial design before any corrections or improvements were applied in program form. As identified during analysis, the design contains several logical errors and oversights.

```
num_zones = int(input("Enter number of zones: "))

num_spaces = int(input("Enter number of parking spaces per zone: "))

most_free_spaces = 0
zone_with_most_free = 0
current_zone = 1

while current_zone <= num_zones:
    occupied_count = 0
    space_count = 1

    print(f"\n--- Zone {current_zone} ---")

    while space_count <= num_spaces:
        status = int(input(f"Enter status for space {space_count} (1=Occupied, 0=Free): "))

        if status == 1:
            occupied_count = occupied_count + 1

        space_count = space_count + 1

    occupancy_percent = (occupied_count / num_spaces)

    print(f"Occupied spaces: {occupied_count}")
    print(f"Percentage occupied: {occupancy_percent}%")
```



```
37
38
39 | if occupancy_percent > 8:
40 |     print("WARNING: Occupancy exceeds 80%!")
41
42
43     free_spaces = num_spaces - occupied_count
44
45
46     if free_spaces > most_free_spaces:
47         most_free_spaces = free_spaces
48         zone_with_most_free = current_zone
49
50     current_zone = current_zone + 1
51
52
53 print("\n--- SUMMARY ---")
54 print(f"Zone {zone_with_most_free} has the most free spaces ({most_free_spaces} spaces).")
55
```

2.2 DETAILED CODE EXPLANATION

Each section of the code is analysed below to explain its intended function, how it functions, and exactly where the design logic fails. This breakdown is critical for understanding the methods observed and used during the testing phase.

Input Section

```
num_zones = int(input("Enter number of zones: "))

num_spaces = int(input("Enter number of parking spaces per zone: "))
```

Function: This section collects the two primary variables required by the system: how many zones the car park contains, and how many spaces exist within each zone.

Operation: The input() function registers the text entered by the user, and the int() function attempts to convert that text into an integer value useable for use in calculations and loops.

Design Flaws

No validation: If the user enters anything other than a whole number — such as letters (aka string), symbols, decimal points, or leaves the prompt blank — the conversion process fails immediately, causing the program to terminate with a runtime error. The original flowchart assumes perfect input, which is unrealistic in practice.



Fixed zone size: The design assumes every zone has **exactly** the same number of spaces. In real-world scenarios, car parks almost always have zones of varying sizes, making this inflexible and limiting practical use.

Variable Initialisation

```
most_free_spaces = 0
zone_with_most_free = 0
current_zone = 1
```

Function: Variables are created and assigned initial values before processing begins. These act as temporary storage to keep track of data as the program runs.

Operation:

- **most_free_spaces** is intended to store the highest number of free spaces found in any zone.
- **zone_with_most_free** stores the identification number of that zone.
- **current_zone** tracks which zone is currently being processed, starting from Zone 1.

Design Flaws

Starting `most_free_spaces` at zero creates a logical error. If all zones are completely full, every zone will have zero free spaces. As a result, the variable will never be updated from its initial value, and the program will incorrectly report Zone 0 as the zone with the most free spaces — a zone that does not exist. This produces a completely incorrect final output.

Outer Zone Loop

```
while current_zone <= num_zones:
    occupied_count = 0
    space_count = 1

    print(f"\n--- Zone {current_zone} ---")
```

Function: This is the primary control structure that makes sure the program processes every zone in sequence.

Operation:

- The loop condition checks whether the current zone number is less than or equal to the total number of zones entered. If true, the code inside runs; if false, the loop ends.



At the start of each iteration, counters are reset to zero or one. This is important — without resetting, `occupied_count`, values from the previous zone would be carried forward, causing major over-counting errors.

- A header message is printed to help the user identify which zone they are entering data for.

Design Quality: This part of the logic is structurally sound and correctly implemented. The concept of resetting counters at the start of each new iteration follows standard programming practice.

Inner Space Loop

```
while space_count <= num_spaces:

    status = int(input(f"Enter status for space {space_count} (1=occupied, 0=Free): "))

    if status == 1:
        occupied_count = occupied_count + 1

    space_count = space_count + 1
```

Function: This nested loop (loop in a loop) processes every individual parking space within the current zone, collecting occupancy status and maintaining a running total of occupied spaces.

Operation:

- The loop runs once for each space, asking the user to enter 1 for occupied or 0 for free.
- If the **input** is exactly 1, the occupied counter is increased by one.
- If the **input** is anything else, the code assumes the space is free and takes no action.
- The space counter is incremented to move to the next position.

Design Flaws

No range checking: The logic only checks if the value is 1. If the user enters 2, 3, -5, or any other integer, it is accepted as valid input and treated the same as entering 0. This means invalid data is silently accepted and recorded incorrectly, leading to false statistics without the user even realising.



No type checking: As with the earlier inputs, entering **non-numeric data** will crash the program instantly. There is no mechanism to handle user error in a good way.

Calculation and Output

```
occupancy_percent = (occupied_count / num_spaces)

print(f"Occupied spaces: {occupied_count}")
print(f"Percentage occupied: {occupancy_percent}%")
```

```
| if occupancy_percent > 8:
    print("WARNING: Occupancy exceeds 80%!")
```

Function: This section calculates occupancy statistics and provides feedback to the user.

Operation:

- The occupied count is divided by the total spaces to determine use.
- Results are printed to the screen.
- A condition checks whether occupancy exceeds a threshold, displaying a warning message if true.

Design Flaws

Mathematical error: Dividing two integers produces a decimal fraction, not a percentage. For example, 2 occupied out of 4 spaces results in 0.5, but mathematically this represents 50%. To display correctly, the result must be multiplied by 100. Without this step, the output shows 0.5% — a value that is factually incorrect and misleading.

Incorrect threshold: The condition checks whether the value is greater than 8, but the message refers to exceeding 80%. Combined with the previous error, the program compares decimals **against** whole numbers, meaning the warning *will almost never appear even when zones are completely full*.



Comparison Logic

```
if free_spaces > most_free_spaces:  
    most_free_spaces = free_spaces  
    zone_with_most_free = current_zone  
  
current_zone = current_zone + 1
```

Function: This determines which zone has the most available spaces, allowing users to quickly identify the least crowded area.

Operation:

- Free spaces are calculated by subtracting occupied spaces from the total.
- If the current zone has more free spaces than the highest value recorded so far, the stored values are updated.

Design Flaws

As previously noted, starting **most_free_spaces** at zero causes incorrect results when all zones are full.

There is no mechanism to handle ties — if two zones have **exactly** the same number of free spaces, only the first one encountered will be recorded, even if the later zone is identical.

2.3 TESTING METHODOLOGY

To fully evaluate the behaviour of the system, four specific test cases were designed according to the assessment instructions. These cases were chosen to cover normal operation, boundary conditions, and invalid inputs. By applying systematic testing, it is possible to observe exactly how the program responds in different scenarios and identify the underlying causes of any failures.



Test Case A

Inputs:

- Zones: 2
- Spaces: 4
- Zone 1: 1, 1, 0, 0
- Zone 2: 0, 0, 1, 1

Observed Output:

Purpose:

Normal, balanced operation.

Expected 50% occupancy

in both zones.

```
Enter number of zones: 2
Enter number of parking spaces per zone: 4

--- Zone 1 ---
Enter status for space 1 (1=Occupied, 0=Free): 1
Enter status for space 2 (1=Occupied, 0=Free): 1
Enter status for space 3 (1=Occupied, 0=Free): 0
Enter status for space 4 (1=Occupied, 0=Free): 0
Occupied spaces: 2
Percentage occupied: 0.5%

--- Zone 2 ---
Enter status for space 1 (1=Occupied, 0=Free): 0
Enter status for space 2 (1=Occupied, 0=Free): 0
Enter status for space 3 (1=Occupied, 0=Free): 1
Enter status for space 4 (1=Occupied, 0=Free): 1
Occupied spaces: 2
Percentage occupied: 0.5%

--- SUMMARY ---
Zone 1 has the most free spaces (2 spaces).

Process finished with exit code 0
```

Test Case B

Inputs:

- Zones: 2
- Spaces: 3
- Zone 1: 1, 1, 1
- Zone 2: 1, 1, 1

Observed Output:

Purpose:

Boundary condition —

testing behaviour when zones are 100% full.

```
Enter number of zones: 2
Enter number of parking spaces per zone: 3

--- Zone 1 ---
Enter status for space 1 (1=Occupied, 0=Free): 1
Enter status for space 2 (1=Occupied, 0=Free): 1
Enter status for space 3 (1=Occupied, 0=Free): 1
Occupied spaces: 3
Percentage occupied: 1.0%

--- Zone 2 ---
Enter status for space 1 (1=Occupied, 0=Free): 1
Enter status for space 2 (1=Occupied, 0=Free): 1
Enter status for space 3 (1=Occupied, 0=Free): 1
Occupied spaces: 3
Percentage occupied: 1.0%

--- SUMMARY ---
Zone 0 has the most free spaces (0 spaces).

Process finished with exit code 0
```



Test Case C

Inputs:

- Zones: 1
- Spaces: 4
- Zone 1: 1, 2, 0, %

Observed Output:

Purpose:

Invalid input —

testing response to numbers outside

the allowed range and non-numeric characters.

```
Enter number of zones: 1
Enter number of parking spaces per zone: 4

--- Zone 1 ---
Enter status for space 1 (1=Occupied, 0=Free): 1
Enter status for space 2 (1=Occupied, 0=Free): 2
Enter status for space 3 (1=Occupied, 0=Free): 0
Enter status for space 4 (1=Occupied, 0=Free): %
Traceback (most recent call last):
  File "C:\Users\Owner\PyCharmMiscProject\script1.py", line 24, in <module>
    status = int(input(f"Enter status for space {space_count} (1=Occupied, 0=Free): "))
ValueError: invalid literal for int() with base 10: '%'
Process finished with exit code 1
```

Test Case D

Inputs:

- Zones: 2
- Spaces: 3
- Zone 1: 1, 0, 1
- Zone 2: 0, 0, 1

Observed Output:

Purpose:

Mixed occupancy —

testing calculation accuracy

and comparison logic.

```
Enter number of zones: 2
Enter number of parking spaces per zone: 3

--- Zone 1 ---
Enter status for space 1 (1=Occupied, 0=Free): 1
Enter status for space 2 (1=Occupied, 0=Free): 0
Enter status for space 3 (1=Occupied, 0=Free): 1
Occupied spaces: 2
Percentage occupied: 0.6666666666666666%

--- Zone 2 ---
Enter status for space 1 (1=Occupied, 0=Free): 0
Enter status for space 2 (1=Occupied, 0=Free): 0
Enter status for space 3 (1=Occupied, 0=Free): 1
Occupied spaces: 1
Percentage occupied: 0.3333333333333333%

--- SUMMARY ---
Zone 2 has the most free spaces (2 spaces).
Process finished with exit code 0
```



Each test was executed exactly as specified, and the outputs were recorded for analysis.

2.4 OBSERVATION ANALYSIS

Test Case	Observed Output	Error Type	Explanation
A	Percentages shown as decimals (e.g. 0.5%) instead of 50%. No warnings displayed.	Logical / Calculation Error	The formula (occupied / spaces) produces a fraction. To get a percentage, the result must be multiplied by 100. Additionally, the threshold check was written as >8 instead of >80, so warnings were never triggered.
B	Zones are 100% full, but no warning message appeared.	Logical / Condition Error	Due to the errors above, the system saw occupancy as 1.0% rather than 100%, so it never met the incorrect threshold condition to display a warning.
C	Program crashed when entering % and accepted 2 as valid.	Runtime / Validation Error	The flowchart did not include logic to validate inputs. It assumed the user would only type 0 or 1. Any other number or character was either accepted incorrectly or caused the program to crash.



Test Case	Observed Output	Error Type	Explanation
D	Output mathematically correct but displayed poorly. No warnings.	Logical / Formatting Error	While the counting logic worked here, the results were hard to read and the core logic for warnings and percentages remained broken as described in Case A.

3. CORRECTED PROGRAM AND RE-TESTING

3.1 IMPLEMENTATION OF CORRECT CODE

Having identified the logical errors of the flowchart in program form. They can now be rectified, and the program can be completed.

```

num_zones = int(input("Enter number of zones: "))

num_spaces = int(input("Enter number of parking spaces per zone: "))

# Variables for tracking
most_free_spaces = -1
zone_with_most_free = 0

for zone_id in range(1, num_zones + 1):
    occupied_count = 0
    print(f"\n===== PROCESSING ZONE {zone_id} =====")

    for space in range(1, num_spaces + 1):

        while True:
            try:
                status = int(input(f"Space {space} Status (1=Occupied, 0=Free): "))
                if status == 0 or status == 1:
                    break
            except ValueError:
                print("!!! Invalid input. Please enter ONLY 0 or 1.")

        if status == 1:
            occupied_count += 1

    occupancy_percent = (occupied_count / num_spaces) * 100
    free_spaces = num_spaces - occupied_count

    print(f"Occupied Spaces: {occupied_count} / {num_spaces}")
    print(f"Occupancy Rate: {occupancy_percent:.2f}%")

```



```

39
40
41     if occupancy_percent > 80:
42         print("⚠ WARNING: Occupancy exceeds 80% - Zone nearing capacity!")
43
44
45     if free_spaces > most_free_spaces:
46         most_free_spaces = free_spaces
47         zone_with_most_free = zone_id
48
49
50 print("\n==== FINAL SUMMARY =====")
51 print(f"Zone {zone_with_most_free} currently has the MOST FREE SPACES ({most_free_spaces} spaces available).")
52 print("=====")
53

```

3.2 RE-TESTING AND VERIFICATION

Test Case	Observed Output	Was the issue resolved?	Explanation
A 2 Zones, 4 Spaces Z1: 1,1,0,0 Z2: 0,0,1,1	Occupancy Rate: 50.00% Occupancy Rate: 50.00% Zone 1 has the most free spaces (2).	YES	Percentages are now correctly calculated by multiplying by 100 and formatted clearly. No warning appears as expected, since 50% is below the 80% threshold.
B 2 Zones, 3 Spaces Z1: 1,1,1 Z2: 1,1,1	Occupancy Rate: 100.00% WARNING: Occupancy exceeds 80% Occupancy Rate: 100.00%	YES	Threshold condition corrected to >80. Warnings appear correctly. Output is clear and accurate.



Test Case	Observed Output	Was the issue resolved?	Explanation
	WARNING: Occupancy exceeds 80% Zone 1 has the most free spaces (0).		
C 1 Zone, 4 Spaces Inputs: 1, 2, 0, %	Input '2': "!! Invalid input. Please enter ONLY 0 or 1." Input '%': "!! Invalid character. Please enter a NUMBER..." The program continues running and asks again until valid data is entered.	YES	Added while True loops and try...except blocks. This filters invalid numbers and characters, preventing crashes and ensuring data integrity.
D 2 Zones, 3 Spaces Z1: 1,0,1 Z2: 0,0,1	Occupancy Rate: 66.67% Occupancy Rate: 33.33% Zone 2 has the most free spaces (2).	YES	Logic works perfectly. Calculations are accurate, formatting is clean, and since neither zone exceeds 80%, no warning appears. Comparison logic identifies Zone 2 correctly.



Test Case A

Inputs:

Zones: 2

Spaces: 4

Zone 1: 1, 1, 0, 0

Zone 2: 0, 0, 1, 1

Observations:

- The program correctly counted the number of occupied spaces.
- However, the percentage displayed was mathematically incorrect — instead of 50%, it showed 0.5%.
- No warning messages were displayed, even though the system appeared to be functioning otherwise.
- The comparison logic correctly identified Zone 1 as having the same number of free spaces as Zone 2, but Zone 1 was haphazardly chosen as the result.

```
Enter number of zones: 2
Enter number of parking spaces per zone: 4

===== PROCESSING ZONE 1 =====
Space 1 Status (1=Occupied, 0=Free): 1
Space 2 Status (1=Occupied, 0=Free): 1
Space 3 Status (1=Occupied, 0=Free): 0
Space 4 Status (1=Occupied, 0=Free): 0
Occupied Spaces: 2 / 4
Occupancy Rate: 50.00%

===== PROCESSING ZONE 2 =====
Space 1 Status (1=Occupied, 0=Free): 0
Space 2 Status (1=Occupied, 0=Free): 0
Space 3 Status (1=Occupied, 0=Free): 1
Space 4 Status (1=Occupied, 0=Free): 1
Occupied Spaces: 2 / 4
Occupancy Rate: 50.00%

===== FINAL SUMMARY =====
Zone 1 currently has the MOST FREE SPACES (2 spaces available).
=====
Process finished with exit code 0
```

Test Case B

Inputs:

Zones: 2

Spaces: 3

Zone 1: 1, 1, 1

Zone 2: 1, 1, 1

Observations

- Both zones were correctly recorded as fully occupied.
- Percentage values were displayed as 1.0% instead of 100%, confirming the calculation error.

```
Enter number of zones: 2
Enter number of parking spaces per zone: 3

===== PROCESSING ZONE 1 =====
Space 1 Status (1=Occupied, 0=Free): 1
Space 2 Status (1=Occupied, 0=Free): 1
Space 3 Status (1=Occupied, 0=Free): 1
Occupied Spaces: 3 / 3
Occupancy Rate: 100.00%
▲ WARNING: Occupancy exceeds 80% - Zone nearing capacity!

===== PROCESSING ZONE 2 =====
Space 1 Status (1=Occupied, 0=Free): 1
Space 2 Status (1=Occupied, 0=Free): 1
Space 3 Status (1=Occupied, 0=Free): 1
Occupied Spaces: 3 / 3
Occupancy Rate: 100.00%
▲ WARNING: Occupancy exceeds 80% - Zone nearing capacity!

===== FINAL SUMMARY =====
Zone 1 currently has the MOST FREE SPACES (0 spaces available).
=====
Process finished with exit code 0
```



- Importantly, no warning messages appeared, despite both zones being completely full and exceeding the intended 80% threshold. This shows clearly that the warning logic is broken.

- The summary reported Zone 1 as having the most free spaces, but the value shown was zero. While mathematically correct, this is not useful information for the user and could be misleading.

Test Case C

Inputs:

Zones: 1

Spaces: 4

Zone 1: 1, 2, 0, %

Observations

When entering 2, the program accepted it without complaint and treated it the same as entering 0. No error message was shown to the user.

When entering the symbol %, the program crashed immediately. The execution terminated and no further data could be entered or processed.

This confirms that there is absolutely no protection against invalid input, making the system highly unstable.

Test Case D

Inputs:

Zones: 2

Spaces: 3

Zone 1: 1, 0, 1

Zone 2: 0, 0, 1

Observations

- The counting logic worked correctly — 2 occupied spaces were recorded for Zone 1 and 1 for Zone 2.

```
Enter number of zones: 1
Enter number of parking spaces per zone: 4

===== PROCESSING ZONE 1 =====
Space 1 Status (1=Occupied, 0=Free): 1
Space 2 Status (1=Occupied, 0=Free): 2
!! Invalid input. Please enter ONLY 0 or 1.
Space 2 Status (1=Occupied, 0=Free): 0
Space 3 Status (1=Occupied, 0=Free): %
!! Invalid character. Please enter a NUMBER (0 or 1).
Space 3 Status (1=Occupied, 0=Free):
```

```
Enter number of zones: 2
Enter number of parking spaces per zone: 3

===== PROCESSING ZONE 1 =====
Space 1 Status (1=Occupied, 0=Free): 1
Space 2 Status (1=Occupied, 0=Free): 0
Space 3 Status (1=Occupied, 0=Free): 1
Occupied Spaces: 2 / 3
Occupancy Rate: 66.67%

===== PROCESSING ZONE 2 =====
Space 1 Status (1=Occupied, 0=Free): 0
Space 2 Status (1=Occupied, 0=Free): 0
Space 3 Status (1=Occupied, 0=Free): 1
Occupied Spaces: 1 / 3
Occupancy Rate: 33.33%

===== FINAL SUMMARY =====
Zone 2 currently has the MOST FREE SPACES (2 spaces available).
=====

Process finished with exit code 0
```



- However, the percentage values were displayed as long decimal numbers such as 0.666...% instead of the correct 66.67%. This makes the output confusing and difficult to interpret.

- No warning messages were displayed, even though Zone 1 was 66% occupied — close to the intended threshold but not quite reaching it. This shows the warning logic was simply never going to trigger correctly due to the earlier mathematical error.

- The comparison logic correctly identified Zone 2 as having more free spaces, so this part of the design was functioning as intended despite the other errors.

Through this analysis, it becomes clear that the original design suffered from **systemic flaws** rather than isolated mistakes. The errors affected data input, mathematical processing, logical decision-making, and user output. This highlights the importance of thorough testing, as issues that appear small on paper can render a system completely unusable in practice.

4. IMPROVING LOGIC AND EFFICIENCY

4.1 DIFFICULTIES FACED WITHOUT ADVANCED DATA STRUCTURES

Without the use of advanced data structures, the original flowchart design relied entirely on simple variables and loops. Created various significant limitations for what could be done, Such as:

1. **Data Loss:** As soon as the program finished processing one zone, the values were overwritten by the next zone's data. There was no way to recall previous results or compare multiple zones easily at the end.
2. **Manual Management:** Counters had to be manually reset to zero at the start of every zone loop. If this step was missed in the logic, values carried over, causing major calculation errors.
3. **Limited Scalability:** Adding new features such as sorting zones or generating reports required rewriting large sections of code because there was no central place to store all the information.



How this affected design and efficiency

The program became **linear and rigid**. It worked step-by-step and could not easily adapt if requirements changed. Debugging was harder because errors in one part of the process affected later parts unpredictably.

Recommended Improvements

To make the system more efficient and professional, **data structures such as Lists or Dictionaries** should be used [1].

- A dictionary could store each zone's data as a record, e.g. `zone_data = {1: {'occupied': 2, 'total': 4}, 2: {'occupied': 1, 'total': 4}}`
- This keeps all data in memory permanently.
- It allows the use of built-in functions to sort, filter or analyse data instantly.
- This reduces code repetition, improves readability, and makes the program much easier to update or expand in future.

5. REFLECTION ON LEARNING

5.1 MOST CHALLENGING PROGRAMMING CONCEPTS

Most challenging programming concepts

The concepts that proved to be the biggest challenge to overcome were **Input Validation** and **Error Handling**. Writing code that works when everything goes right is relatively straightforward, but ensuring the program behaves safely when users make mistakes requires careful planning and use of logic. Understanding how to nest loops so the program keeps asking until valid data is entered was initially difficult to visualise but achievable after some research and trial and error.

Another challenge was understanding **variable scope** challenging – specifically realising when a variable retains its value or is reset, and how this affects the variable calculations across different parts of the program/code.

5.2 HOW WERE THE CHALLENGES OVERCOME

The difficulties encountered in this task were overcome by:



1. **Planning with Flowcharts:** Before writing code, I drew diagrams showing exactly what should happen for both valid and invalid inputs [2]. This helped me structure the logic clearly to be later used within the program.
2. **Modular Testing:** Instead of writing the whole program in a single go, small test sections were written over time. For example, the validation loop was written and tested several times before being added to the main system.
3. **Debugging:** The author learned to use print statements to check variable values at every step. This helped me identify when data was being lost or carried over incorrectly within most if not all, sections of the program.

5.3 APPLICATION IN THE TASK

- **Input Validation:** Was applied specifically in **Test Case C**. The original design crashed, but by using a while True loop and try...except blocks in the corrected version, the program now safely handles all types of incorrect input.
- **Condition Logic:** Applied in **Test Cases A and B**. Careful observation had to be done when checking the mathematical formula and the threshold condition to ensure warnings appeared exactly when necessary.
- **Comparison Logic:** Applied across all zones to ensure the correct zone was identified as having the most free spaces.

5.4 MOST USEFUL TEST CASE

Test Case C was the most valuable. It clearly exposed the limitations of the original design. Seeing the program crash when entering simple characters highlighted just how important validation and error handling are in real-world software. It showed that a program is not just about doing the right thing, but also about *not doing the wrong thing* when things go wrong.

5.5 IMPACT OF TESTING ON UNDERSTANDING

Running the test cases showed that the flowcharts are just abstract models – they often look correct on paper but contain logical gaps when translated into real code. Testing bridges the gap between theory and practical application. It helped in the process of understanding that programming is a cycle: design → implement → test → fix. It is not enough to just follow the diagram; you must validate behaviour under different conditions to ensure reliability & efficiency in whatever program is being constructed. Whether it be for a Smart Parking system or a script for an entity in a video game. The



cycle of design → implement → test → fix is a foundational aspect of the programming experience.



6. BIBLIOGRAPHY

REFERENCES

- [1] Python Software Foundation. <https://docs.python.org/3/tutorial/datastructures.html> . 2023.
- [2] Zelle, J. " Python Programming: An Introduction to Computer Science. ." M.Zelle, John. 3rd ed. Franklin, Beedle & Associates, 20, n.d.



FINAL GRADE

GENERAL COMMENTS

42/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

GRADING FORM: ACCA4020 RUBRIC 2026

ROLINHLANHLA MOYO

42

TASK 1A [20]

Correct implementation of flowchart (even with errors) [8] Code runs (even if incorrect output) [5] All test cases attempted [4] Screenshots/evidence provided [3]



Basic implementation is present with some test cases executed. However, not all errors are clearly demonstrated, and evidence of program behaviour is limited or unclear. Most of the cases reported are very generic

10

TASK 1B [15]

Correct outputs recorded [5] Identification of error types [5] Clear explanation of errors [5]



Limited or incorrect identification of errors. Key issues such as syntax errors, logical flaws, and

6

validation problems are not clearly recognised

TASK 2A [20]

Errors correctly fixed [10] Improved logic (loops, conditions) [6] Input validation added [4]



Some corrections have been made, but key issues remain, particularly in validation or logical comparison between zones

9

TASK 2B [20]

All test cases re-run [6] Outputs clearly recorded [6] Comparison with old results [4] Explanation of fixes [4]



Some re-testing is completed, but not all issues are fully resolved or clearly demonstrated. Explanation of fixes is limited

8

TASK 3 [10]

Understanding of difficulty [5] Impact on design/efficiency [5]



Explanation is very generic; it does not reflect the true challenges/difficulties

4

TASK 4 [15]

Identifies challenges [5] Examples from assignment [5] Reflection on testing/learning [5]



Reflection lacks depth and does not clearly link to the assignment tasks

5