

8/10/2026

Scripting and Programming Concepts

Smart Home Device Manager
System for Bright Minds Academy

Module Name: Scripting and Programming
Concepts

Module Code: ACCA 5036

Student Name: Evgeni Veselinov Iliev

Student ID: 2424555

Lecturer's Name: Koushik Modak

Code Link:

https://github.com/2424555/assessment_code_A

Table of Contents

1. Analysis	2
1.1 Introduction	2
1.2 Abstraction and Problem Decomposition.....	2
1.3 Requirements Analysis.....	3
1.4 Feasibility and Scope.....	4
2. Software Design and Development	5
2.1 Development Approach (SDLC).....	5
2.2 Design.....	5
2.2.1 Flowcharts.....	5
2.2.2 Pseudocode.....	10
2.3 Implementation Decisions	13
2.3.1 Data Structures and Variables	13
2.3.2 Programming Constructs	14
2.3.3 Python Modules and Libraries	14
2.3.4 Code Organisation and Good Practice	15
2.4 Source Code	18
3. Testing and Debugging Report.....	18
3.1 Importance of Programme Testing.....	18
3.2 Types of Programming Errors and Debugging	18
3.3 Test Cases.....	19
3.4 Testing Evidence	20
3.5 Debugging Reflection	26
4. Evaluation	26
4.1 Challenges Faced.....	26
4.2 Future Improvements	27
4.3 Self-Reflection	27
5. Conclusion.....	28
References	29
Appendix A - Short User Guide	30

1. Analysis

1.1 Introduction

Bright Minds Academy is a small private tutoring centre that is introducing smart devices into its classrooms. The devices include smart lights, thermostats and projectors. At the moment, staff manage these devices manually, which makes it difficult to know which devices have been registered, where they are located and whether they are online, offline or unavailable because of maintenance. The main need is therefore a simple computer-based system that stores the important details in one place and allows staff to find the information quickly. The proposed solution is a menu-based Python program called Smart Device Manager. It will allow staff to add a device, update its status, view the full device list and search for a device by name. The system will not attempt to control real smart devices or connect to a live network because these functions are outside the scope of the project.

1.2 Abstraction and Problem Decomposition

I used abstraction to keep the real smart-classroom problem at a manageable level. A real projector or thermostat may have an IP address, serial number, model, firmware version and many technical settings, but most of these details are not needed for this coursework system. Each device is therefore represented using only three main fields: device name, room/location and current status. This reduces unnecessary data entry and makes the information easier for staff to read. I also used problem decomposition by splitting the program into smaller tasks: adding a device, updating a status, viewing all devices, searching by name, validating input and saving/loading data. Each task can then be handled by a separate Python function, which makes the program easier to understand, test and change later.

1.3 Requirements Analysis

Requirements analysis is important because it defines what the software should do before the coding starts [1]. For this project, I treated the four functions in the brief as the essential requirements. Input validation is also needed because blank names, duplicate devices or invalid status choices would make the records less reliable. I included a small JSON file so the device list can still be available after the program closes. This does not replace the list of dictionaries used by the program; it only saves the same data between sessions. I decided not to include remote control, automatic network discovery, staff accounts or a graphical dashboard in the first version because they are outside the main problem. To keep the scope clear, I used MoSCoW prioritisation to separate Must, Should, Could and Won't Have requirements [2].

Table 1. Functional Requirements

ID	Functional requirement	Description
FR1	Add new device	Staff can enter a unique device name and the room/location.
FR2	Update device status	Staff can set a device to Online, Offline or Under Maintenance.
FR3	View device list	The system displays all device names, locations and current statuses.
FR4	Simple search	Staff can type a device name and see its location and status.
FR5	Input validation	The system rejects blank values, duplicate names and invalid menu/status choices.
FR6	Exit and save	The user can close the program safely and saved data is kept for the next session.

Table 2. Non-Functional Requirements

Non-functional requirement	How it is addressed
Reliability	Records should remain consistent and invalid values should not be accepted.
Performance	Adding, viewing and searching a small device list should respond immediately on a normal classroom computer.
Security / data safety	Device records should be saved to a local file to reduce accidental loss when the program closes.
Accessibility	The console menu should use clear wording, numbered choices and readable messages for non-technical staff.
Scalability	The structure should allow more devices and extra fields to be added later without redesigning the whole program.

Table 3. MoSCoW Prioritisation of Requirements

Priority	Features
Must Have	Add device; update status; view list; search by name; validation
Should Have	Save and reload device data using JSON
Could Have	Edit/delete devices; sorting; reports; graphical interface
Won't Have in this version	Live network discovery; remote control; mobile application

1.4 Feasibility and Scope

The proposed system is feasible for a small tutoring centre because it only requires Python and a local file; no database server, paid package or specialist hardware is needed. The console interface also keeps development time reasonable. However, this design has limits. It is mainly intended for one computer and a small number of records, and it does not check the real network state of a device. The status shown is therefore the status entered by staff rather than live information. This limitation is acceptable for the coursework version because the brief asks for tracking rather than real-time control.

2. Software Design and Development

2.1 Development Approach (SDLC)

I used a simple iterative development approach for this project. The software development life cycle separates work into stages such as requirements, design, coding and testing [3]. Instead of writing the whole program in one attempt, I planned to build and check it in small parts. I started with the data structure and main menu, then added the device-registration function. The update, view and search functions were added afterwards, with validation checked as each part was completed. This approach suited a small Python project because an error could be found and corrected before more features were added.

2.2 Design

2.2.1 Flowcharts

I used standard flowchart symbols to plan the main program and the most important functions before coding. The diagrams show where user input is taken, where decisions are made and how the program returns to the menu.

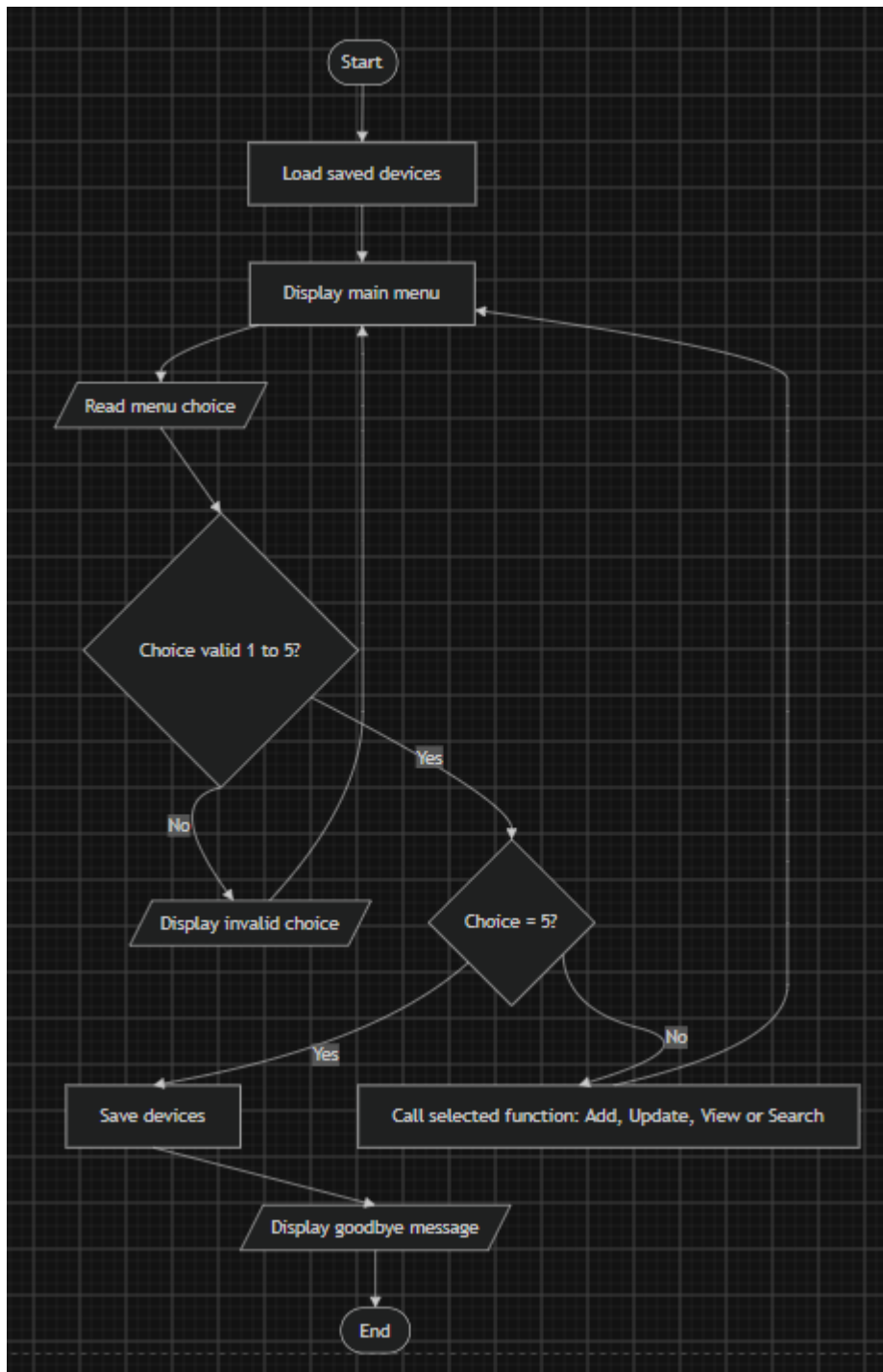


Figure 1.Main program flowchart.

Figure 1 shows the overall menu loop. The program loads saved records, accepts a menu choice and returns to the menu after each function until the user selects Exit.

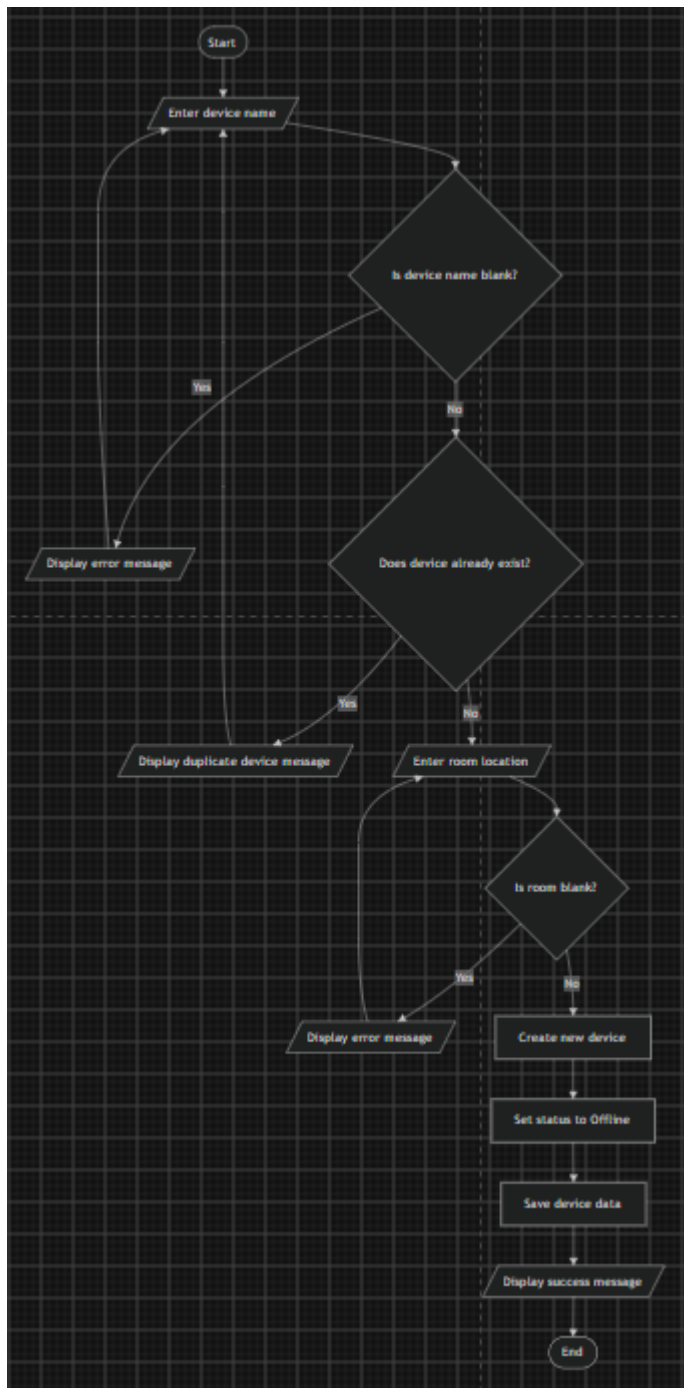


Figure 2.Add device flowchart with validation.

Figure 2 focuses on validation. A blank name or room is rejected, and a duplicate device name is not added to the list.

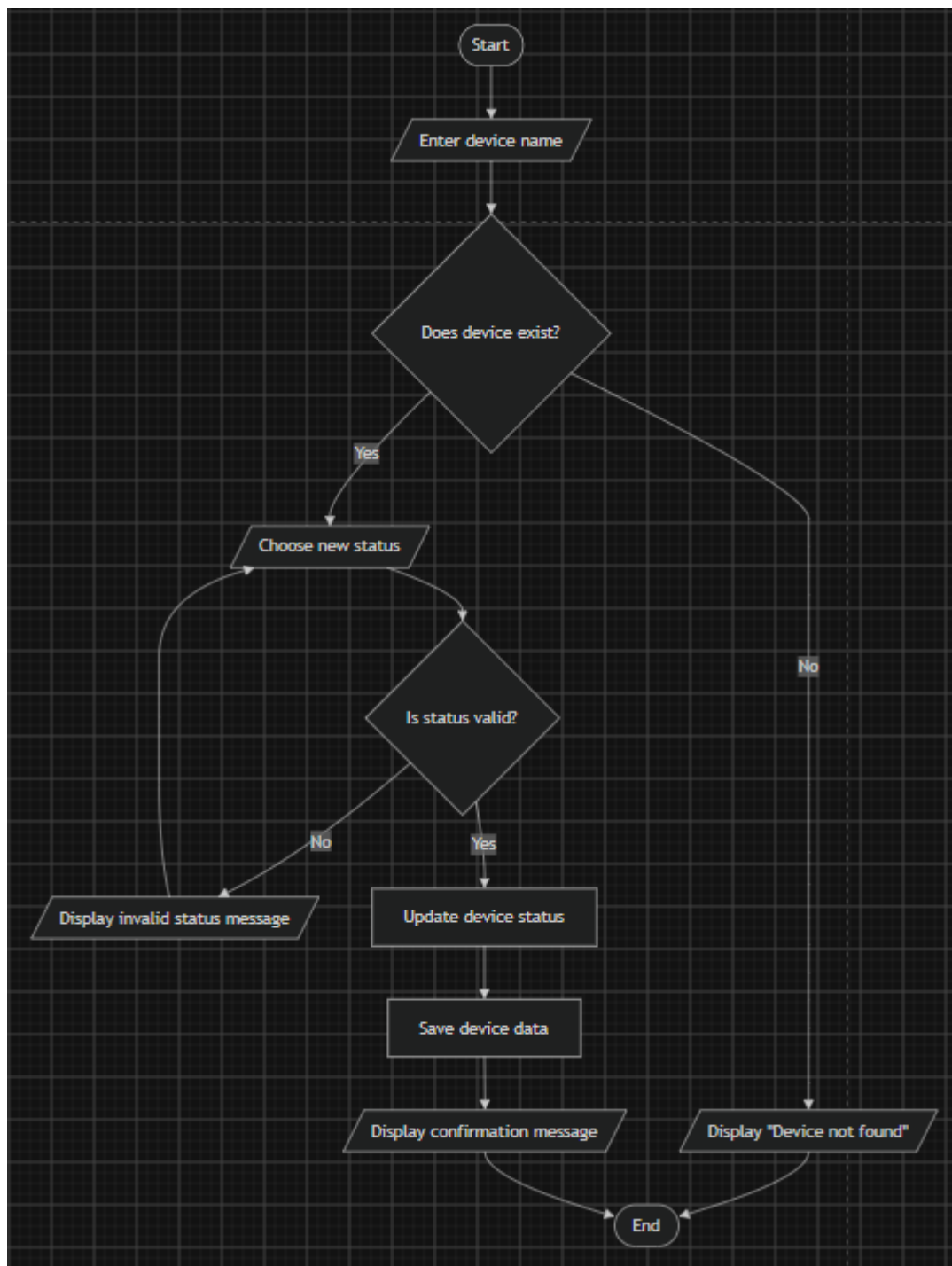


Figure 3. Update device status flowchart.

Figure 3 shows how the program checks that a device exists and then restricts the status choice to Online, Offline or Under Maintenance.

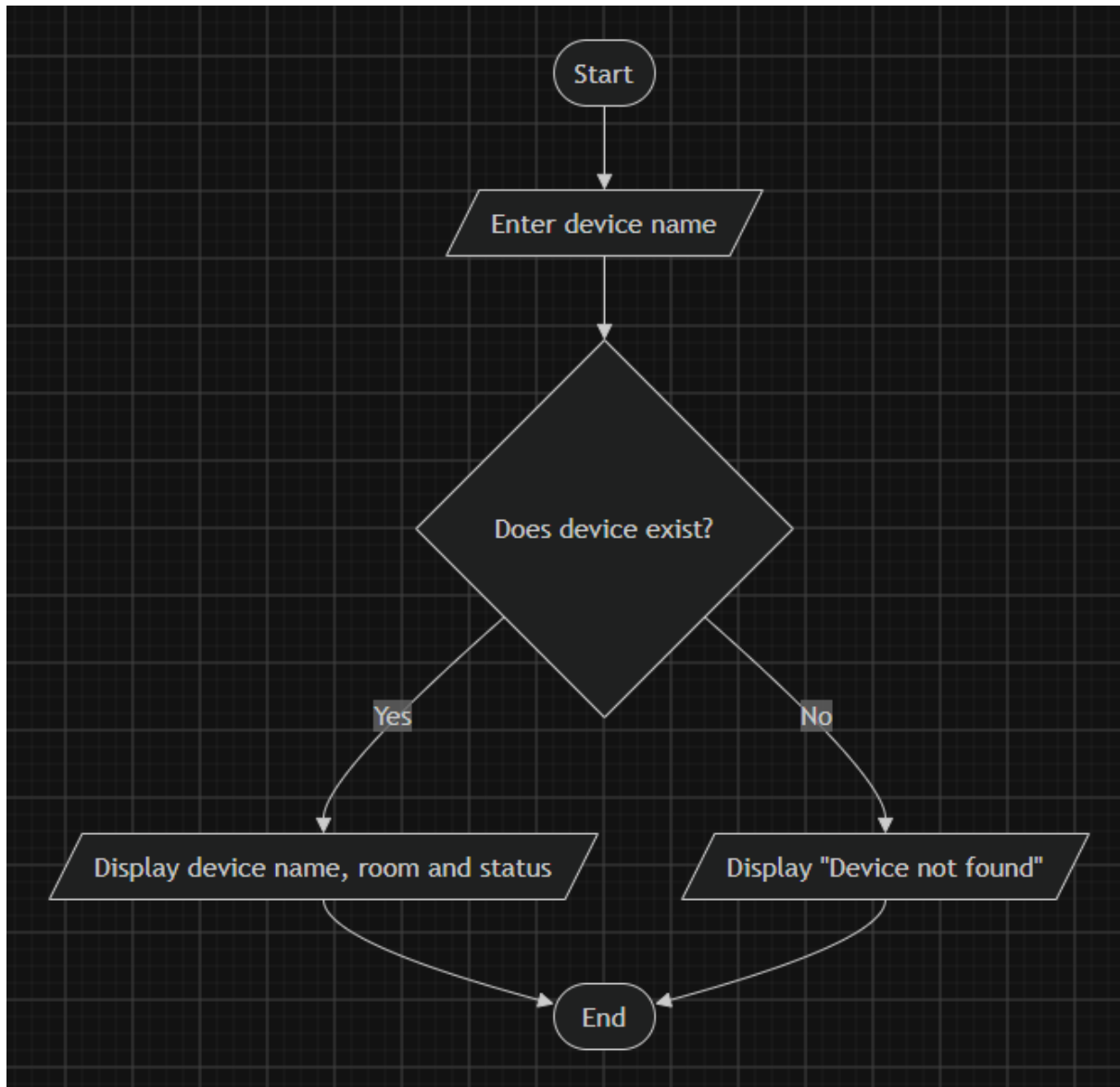


Figure 4. Search device flowchart.

Figure 4 represents the simple search requirement. If the name is found, its room and status are displayed; otherwise, the user receives a clear not-found message.

2.2.2 Pseudocode

The pseudocode below describes the main algorithm and the four core functions before they are written in Python.

```
BEGIN SmartDeviceManager
    devices ← LOAD saved device list

    REPEAT
        DISPLAY main menu
        INPUT choice

        IF choice = 1 THEN
            CALL AddDevice
        ELSE IF choice = 2 THEN
            CALL UpdateStatus
        ELSE IF choice = 3 THEN
            CALL ViewDevices
        ELSE IF choice = 4 THEN
            CALL SearchDevice
        ELSE IF choice = 5 THEN
            SAVE devices
            DISPLAY goodbye message
            STOP
        ELSE
            DISPLAY invalid choice message
        END IF
    FOREVER
END
```

FUNCTION AddDevice(devices)

 INPUT deviceName

 WHILE deviceName is blank

 DISPLAY error message

 INPUT deviceName again

 END WHILE

 IF deviceName already exists THEN

 DISPLAY duplicate message

 RETURN

 END IF

 INPUT room

 WHILE room is blank

 DISPLAY error message

 INPUT room again

 END WHILE

 newDevice ← {name: deviceName, room: room, status: "Offline"}

 APPEND newDevice TO devices

 SAVE devices

 DISPLAY success message

END FUNCTION

FUNCTION UpdateStatus(devices)

 IF devices is empty THEN

 DISPLAY no devices message

 RETURN

 END IF

 INPUT deviceName

 device ← FIND deviceName

```

    IF device does not exist THEN
        DISPLAY device not found
        RETURN
    END IF
REPEAT
    DISPLAY 1 Online, 2 Offline, 3 Under Maintenance
    INPUT statusChoice
    UNTIL statusChoice is valid

    UPDATE device status
    SAVE devices
    DISPLAY confirmation
END FUNCTION

```

```

FUNCTION ViewDevices(devices)
    IF devices is empty THEN
        DISPLAY no devices message
    ELSE
        FOR EACH device IN devices
            DISPLAY device name, room and status
        END FOR
    END IF
END FUNCTION

```

```

FUNCTION SearchDevice(devices)
    INPUT deviceName
    device ← FIND deviceName without case sensitivity
    IF device exists THEN
        DISPLAY device name, room and status
    ELSE
        DISPLAY device not found
    END IF
END FUNCTION

```

2.3 Implementation Decisions

2.3.1 Data Structures and Variables

The main data structure is a list of dictionaries. Python dictionaries store data as key-value pairs [4]. Each dictionary represents one device and contains the keys name, room and status. The dictionaries are stored inside a list because the system needs to hold several device records and loop through them when displaying or searching. This structure is simple and appropriate for the size of the tutoring centre. A database would add unnecessary complexity to the current requirements.

```
devices = [  
    {"name": "Projector B2", "room": "Room 2", "status": "Online"},  
    {"name": "Smart Light A1", "room": "Room 1", "status": "Under Maintenance"}  
]
```

A JSON file is used only for persistence. Python's built-in json module can encode Python data into JSON and decode it again [5]. This means the list of dictionaries can be saved when a change is made and loaded again when the program starts. The system still uses normal Python lists and dictionaries during execution, so the design remains within the project scope.

Table 4. Main Variables Used in the Program

Variable	Type	Purpose
devices	List	Stores every device dictionary in memory.
device_name	String	Stores the name typed by the user.
room	String	Stores the classroom or location.
status	String	Stores Online, Offline or Under Maintenance.
choice	String	Stores a main-menu selection from 1 to 5.
new_status	String	Stores the validated status selected by the user.
device	Dictionary / None	Stores the record returned by find_device(), or None if not found.

2.3.2 Programming Constructs

Selection is implemented using if, elif and else statements. The main menu uses these statements to decide which function to call, while other parts of the program use selection to check for duplicate devices, missing devices and valid status choices. Python provides if statements and loop structures as core control-flow tools [6].

Iteration is used in two ways. A while True loop keeps the main menu running until the user chooses Exit, and validation loops repeat until acceptable input is entered. For loops are used to search through the device list and to display every device. Functions such as add_device(), update_device_status(), view_devices() and search_device() divide the program into manageable parts. Helper functions such as find_device() and get_non_empty_input() prevent the same logic from being repeated.

Error handling is used when loading and saving the JSON file. FileNotFoundError is handled on the first run, when a data file does not yet exist. json.JSONDecodeError is also handled so that damaged JSON does not cause the whole program to terminate. Python distinguishes syntax errors from exceptions that occur during execution, and try/except can be used to handle expected exceptions [7].

2.3.3 Python Modules and Libraries

Table 5. Python Modules and Libraries

Module	Type	Reason for use
json	Built-in Python module	Saves and loads the list of dictionaries in devices.json. It was chosen because no external installation is needed [5].

No third-party Python packages are required. Keeping the program within the standard library makes setup easier on a classroom computer and keeps the solution appropriate for a small pilot system.

2.3.4 Code Organisation and Good Practice

The source code uses descriptive snake_case names, small functions, short comments and docstrings. PEP 8 provides widely used style guidance for Python code, including naming and layout conventions [8]. Constants such as DATA_FILE and VALID_STATUSES are placed at the top of the file, while the main() function is placed near the end. The code is separated by responsibility so a reader can follow the program without moving through one very large block.

```
7
8  import json
9
10 DATA_FILE = "devices.json"
11 VALID_STATUSES = ["Online", "Offline", "Under Maintenance"]
12
13
14 def load_devices(): 1 usage
15     """Load saved devices from the JSON file."""
16     try:
17         with open(DATA_FILE, "r", encoding="utf-8") as file:
18             data = json.load(file)
19             if isinstance(data, list):
20                 return data
21             return []
22     except FileNotFoundError:
23         return []
24     except json.JSONDecodeError:
25         print("Warning: the data file could not be read. Starting with an empty list.")
26         return []
27
28
29 def save_devices(devices): 3+ usages
30     """Save the current device list to the JSON file."""
31     try:
32         with open(DATA_FILE, "w", encoding="utf-8") as file:
33             json.dump(devices, file, indent=4)
34     except OSError:
35         print("Warning: device data could not be saved.")
```

Figure 5. Example of organised code and the JSON persistence functions.

Figure 5 shows the beginning of main.py in PyCharm. It demonstrates the use of a constant data file, the allowed statuses and the two file-handling functions. The complete code is also pasted as text in Section 2.4, as required by the assignment brief.

```

55
56 def add_device(devices : {__iter__, append}): 1 usage
57     """Add a new device and room to the system."""
58     print("\n--- Add New Device ---")
59     device_name = get_non_empty_input("Enter device name: ")
60
61     if find_device(devices, device_name) is not None:
62         print("A device with this name already exists.")
63         return
64
65     room = get_non_empty_input("Enter room/location: ")
66
67     new_device = {
68         "name": device_name.title(),
69         "room": room.title(),
70         "status": "Offline"
71     }
72
73     devices.append(new_device)
74     save_devices(devices)
75     print(f"{new_device['name']} was added successfully with status Offline.")
76
77
78 def choose_status(): 1 usage
79     """Display the status menu and return a valid status."""
80     while True:
81         print("\nChoose the new status:")
82         print("1. Online")
83         print("2. Offline")
84         print("3. Under Maintenance")
85         choice = input("Enter status choice (1-3): ").strip()
86
87         if choice == "1":
88             return VALID_STATUSES[0]
89         if choice == "2":
90             return VALID_STATUSES[1]
91         if choice == "3":
92             return VALID_STATUSES[2]
93
94         print("Invalid status. Please enter 1, 2 or 3.")

```

Figure 6. Core device-management functions in PyCharm.

```

96
97  def update_device_status(devices : {__iter__}): 1 usage
98      """Update the status of an existing device."""
99      print("\n--- Update Device Status ---")
100
101      if not devices:
102          print("No devices have been added yet.")
103          return
104
105      device_name = get_non_empty_input("Enter device name: ")
106      device = find_device(devices, device_name)
107
108      if device is None:
109          print("Device not found.")
110          return
111
112      new_status = choose_status()
113      device["status"] = new_status
114      save_devices(devices)
115      print(f"Status updated: {device['name']} is now {new_status}.")
116
117
118  def view_devices(devices : {__iter__, __len__}): 1 usage
119      """Display all devices, rooms and statuses."""
120      print("\n--- Device List ---")
121
122      if not devices:
123          print("No devices have been added yet.")
124          return
125
126      print(f"{'Device Name':<25}{'Room/Location':<20}{'Status':<20}")
127      print("-" * 65)
128
129      for device in devices:
130          print(f"{device['name']:<25}{device['room']:<20}{device['status']:<20}")
131
132      print(f"\nTotal devices: {len(devices)}")
133
134
135  def search_device(devices : {__iter__}): 1 usage

```

Figure 7. Core device-management functions in PyCharm

Figure 6 and 7 shows how the program uses separate functions, validation and a saved list of dictionaries. The screenshot is supporting evidence only; it does not replace the full source-code listing below.

2.4 Source Code

GitHub repository: https://github.com/2424555/assessment_code_ACCA5036

3. Testing and Debugging Report

3.1 Importance of Programme Testing

Programme testing is important because a program can run without immediately showing every problem. Software testing is recognised as a core area of software engineering practice [9]. In this project, the tests check that the required features give the expected result and that incorrect input is handled safely. Staff may leave a name blank, try to add a duplicate device, type an invalid menu choice or search for a device that is not stored. These situations should give a clear message instead of crashing the program or saving misleading data. I included normal, erroneous and edge cases so the validation can be shown rather than only described.

3.2 Types of Programming Errors and Debugging

A syntax error happens when Python code does not follow the language rules, for example a missing colon after an if statement. Python normally reports the line where the parser found the problem, and the code must be corrected before the program can run. A runtime error happens during execution. An example would be attempting to read damaged JSON without handling the resulting exception. Python documentation explains that unhandled exceptions stop normal execution, while expected exceptions can be caught with try/except [7].

A logical error is different because the program runs but gives the wrong result. For example, if the search compared device names using exact capital letters, a stored device called 'Projector B2' could fail to match the search 'projector b2'. The final program avoids this by converting both values to lower case inside find_device(). A sensible debugging process is to reproduce the problem, identify the function responsible, inspect the values being compared, correct the code and repeat the same test.

3.3 Test Cases

Table 6 includes normal, erroneous and boundary/edge cases. The tests cover both the required functions and the validation used to deal with incorrect input.

Table 6. Test Cases and Results

ID	Test	Input	Expected result	Actual result	Pass
T1	Add device - normal	Smart Light A1 / Room 1	Device is added with default status Offline.	Device added successfully with Offline status.	pass
T2	Add device - blank name	Press Enter for device name	Error message is shown and the program asks again.	Blank device name was rejected, and the user was asked to enter a valid value.	pass
T3	Add device - duplicate	Add Smart Light A1 twice	Second record is rejected as a duplicate.	Duplicate device name was rejected, and no second record was added.	pass
T4	Update status - normal	Smart Light A1 → Under Maintenance	Status changes to Under Maintenance and is saved.	Status changed to Under Maintenance successfully.	pass
T5	Update status - invalid	Enter 8, then 1	8 is rejected; 1 is accepted as Online.	Invalid option 8 was rejected and option 1 was accepted.	pass
T6	Update unknown device	Device name: Device	Device not found message; no crash.	Device not found message was displayed and the program continued normally.	pass
T7	View list - multiple records	At least two stored devices	All names, rooms and statuses are displayed in columns.	All stored devices were displayed with their room locations and current statuses.	pass
T8	View list - empty	Run with no devices stored	No devices have been added yet message.	The program displayed a message stating that no devices had been added.	pass
T9	Search - existing	Search projector b2	Projector B2 details are shown even with different capitalisation.	Projector B2 was found successfully using lower-case search input.	pass
T10	Search - missing	Search Unknown Light	Device not found message.	Device not found message was displayed for an unknown device.	pass
T11	Menu validation	Enter 1A	Invalid values rejected; 3 opens device list.	1A was rejected, while option 3 was accepted successfully.	pass
T12	Persistence	Add a device, exit, rerun, view list	Previously saved device is still available.	The previously saved device was still available after restarting the program.	pass

3.4 Testing Evidence

The following PyCharm screenshots provide evidence for selected tests from Table 6. They show the input entered by the user and the resulting output from the program.

```
=====
      BRIGHT MINDS SMART DEVICE MANAGER
=====
1. Add new device
2. Update device status
3. View device list
4. Search for a device
5. Exit
=====
Enter your choice (1-5): 1

--- Add New Device ---
Enter device name: Smart Light A1
Enter room/location: Room 1
Smart Light A1 was added successfully with status Offline.
```

Figure 8. Testing Evidence T1 – Add device successfully

The test shows that a new device can be added with its room location. The system stores the device with Offline as the default status.

```
=====
      BRIGHT MINDS SMART DEVICE MANAGER
=====
1. Add new device
2. Update device status
3. View device list
4. Search for a device
5. Exit
=====
Enter your choice (1-5):
Invalid choice. Please enter a number from 1 to 5.

=====
      BRIGHT MINDS SMART DEVICE MANAGER
=====
1. Add new device
2. Update device status
3. View device list
4. Search for a device
5. Exit
=====
Enter your choice (1-5):
```

Figure 9. Testing Evidence T2 – Blank device name validation

The test shows that the program rejects an empty device name and asks the user to enter a valid value instead.

```

=====
      BRIGHT MINDS SMART DEVICE MANAGER
=====
1. Add new device
2. Update device status
3. View device list
4. Search for a device
5. Exit
=====
Enter your choice (1-5): 1

--- Add New Device ---
Enter device name: Smart Light A1
A device with this name already exists.

```

Figure 10. Testing Evidence T3 – Duplicate device validation

The test shows that the program does not allow the same device name to be added twice. This helps prevent duplicate records in the system.

```

=====
      BRIGHT MINDS SMART DEVICE MANAGER
=====
1. Add new device
2. Update device status
3. View device list
4. Search for a device
5. Exit
=====
Enter your choice (1-5): 2

--- Update Device Status ---
Enter device name: Smart Light A1

Choose the new status:
1. Online
2. Offline
3. Under Maintenance
Enter status choice (1-3): 3
Status updated: Smart Light A1 is now Under Maintenance.

```

Figure 11. Testing Evidence T4 – Update status to Under Maintenance

The test shows that the status of Smart Light A1 was successfully changed to Under Maintenance. This confirms that the program accepts and stores this status correctly.

```

=====
      BRIGHT MINDS SMART DEVICE MANAGER
=====
1. Add new device
2. Update device status
3. View device list
4. Search for a device
5. Exit
=====
Enter your choice (1-5): 2

--- Update Device Status ---
Enter device name: Smart Light A1

Choose the new status:
1. Online
2. Offline
3. Under Maintenance
Enter status choice (1-3): 7
Invalid status. Please enter 1, 2 or 3.

Choose the new status:
1. Online
2. Offline
3. Under Maintenance
Enter status choice (1-3): 1
Status updated: Smart Light A1 is now Online.

```

Figure 12. Testing Evidence T5 – Invalid status option

The test shows that the program rejects an invalid status choice and asks the user to enter another option. After a valid choice was entered, the device status was updated successfully.

```

=====
      BRIGHT MINDS SMART DEVICE MANAGER
=====
1. Add new device
2. Update device status
3. View device list
4. Search for a device
5. Exit
=====
Enter your choice (1-5): 2

--- Update Device Status ---
Enter device name: Device
Device not found.

=====
      BRIGHT MINDS SMART DEVICE MANAGER
=====
1. Add new device
2. Update device status
3. View device list
4. Search for a device
5. Exit
=====
Enter your choice (1-5):

```

Figure 13. Testing Evidence T6 – Update unknown device

The test shows that the program handles an unknown device correctly. When a device name that is not stored in the system was entered, a clear “Device not found” message was displayed and the program continued without crashing.

```
=====
      BRIGHT MINDS SMART DEVICE MANAGER
=====
1. Add new device
2. Update device status
3. View device list
4. Search for a device
5. Exit
=====
Enter your choice (1-5): 3

--- Device List ---
Device Name           Room/Location        Status
-----
Smart Light A1        Room 1              Online
Projector B2          Room 2              Offline

Total devices: 2
```

Figure 14. Testing Evidence T7 - View multiple devices

The test shows that the program can display multiple stored devices together with their room locations and current statuses. This allows staff to view the main information about all registered devices in one place.

```
=====
      BRIGHT MINDS SMART DEVICE MANAGER
=====
1. Add new device
2. Update device status
3. View device list
4. Search for a device
5. Exit
=====
Enter your choice (1-5): 3

--- Device List ---
No devices have been added yet.
```

Figure 15. Testing Evidence T8 – View an empty device list

The test shows that the program handles an empty device list correctly. When no device records are stored, the system displays a clear message instead of producing an error.

```
=====
      BRIGHT MINDS SMART DEVICE MANAGER
=====
1. Add new device
2. Update device status
3. View device list
4. Search for a device
5. Exit
=====
Enter your choice (1-5): 4

--- Search Device ---
Enter device name to search: projector b2
Device name: Projector B2
Room/location: Room 2
Status: Offline
```

Figure 16. Testing Evidence T9 – Search for an existing device

The test shows that the search function can find an existing device even when the name is entered using different capitalisation. The program then displays the correct room and current status.

```
=====
      BRIGHT MINDS SMART DEVICE MANAGER
=====
1. Add new device
2. Update device status
3. View device list
4. Search for a device
5. Exit
=====
Enter your choice (1-5): 4

--- Search Device ---
Enter device name to search: Unknown Light
Device not found.
```

Figure 17. Testing Evidence T10 – Search for a missing device

The test shows that the program handles a search for a device that is not stored in the system. A clear “Device not found” message is displayed instead of causing an error.

```

=====
Enter your choice (1-5): 1A
Invalid choice. Please enter a number from 1 to 5.

=====
          BRIGHT MINDS SMART DEVICE MANAGER
=====
1. Add new device
2. Update device status
3. View device list
4. Search for a device
5. Exit
=====
Enter your choice (1-5): 1

--- Add New Device ---
Enter device name:

```

Figure 18. Testing Evidence T11 - Invalid main menu input

The test shows that the main menu rejects invalid entries. After a valid option was entered, the program continued normally

```

=====
          BRIGHT MINDS SMART DEVICE MANAGER
=====
1. Add new device
2. Update device status
3. View device list
4. Search for a device
5. Exit
=====
Enter your choice (1-5): 5
Device records saved. Goodbye!

```

Figure 19. Testing Evidence T12 – Data persistence after restart

The test confirms that device information is saved successfully and remains available after the program is closed and started again. This shows that the JSON file is being used correctly for persistent storage.

3.5 Debugging Reflection

During testing, I noticed that the room for Projector B2 was displayed as “Room 2 Room 2” instead of “Room 2”. This happened because I accidentally entered the room twice, and the program accepted the value because the validation only checks that the field is not empty. The incorrect value was then saved in the `devices.json` file. I opened the file in PyCharm, corrected the room to “Room 2” and ran the program again to confirm that the correct value was displayed.

This was not a syntax error in the Python code, but it highlighted a limitation in the current validation. The system can prevent blank values, but it cannot always recognise when a user has entered incorrect information. This helped me understand that debugging also involves checking stored data and the final output, not only fixing errors that cause the program to crash. In a future version, stronger validation could be added for room names and other user inputs.

4. Evaluation

4.1 Challenges Faced

One challenge was deciding how much functionality to include. I first considered extra options such as logins, editing devices and more advanced filtering, but this would have made the program larger without improving the four main requirements. I therefore kept the menu focused on adding, updating, viewing and searching. Validation was another challenge because the same type of mistake can happen in different parts of the program. Using small helper functions made the checking more consistent. I also wanted the records to remain after the program closed. JSON added some file-handling code, but I felt this was a useful addition because otherwise staff would have to enter the same devices every time they started the program.

Another design decision was how devices should be searched. The current program uses a linear search, meaning that it checks the device records one at a time. I considered this suitable for Bright Minds Academy because the pilot is expected to contain only a small number of devices. However, this approach would become less efficient if the system had hundreds or thousands of records. For a larger system, a dictionary indexed by device name or a database would provide a more scalable solution.

4.2 Future Improvements

The current system is suitable for a small pilot, but it has clear limitations. A future version could allow staff to edit or delete a device, record the date when a device was last used, sort devices by room and produce simple maintenance reports. A graphical interface could make the system easier to use than a console menu. If the number of devices became much larger, the JSON file could be replaced by a database so that records can be managed more efficiently. JSON was appropriate for this project because it is simple and does not require a database server. However, it would be less suitable if several staff members needed to update the device records at the same time. A database would be more suitable for larger amounts of data and multiple users. Security could also be improved by adding staff accounts and permissions. A more advanced future improvement could be automatic network discovery, but this would move the project from a tracking system towards a real smart-device management platform and would require additional security and networking work.

4.3 Self-Reflection

The project helped me see the connection between planning and actual code. The pseudocode and flowcharts made the functions easier to write because I already knew what each part was supposed to do. I also became more confident with lists and dictionaries, especially with storing several values for one device and then searching through the records. The biggest lesson for me was validation. A program is not really finished just because it works with correct input; it also needs to respond properly when the user makes a mistake. When I do a similar project again, I would keep a short testing and bug log while I code instead of trying to remember problems at the end. That would make the debugging evidence more accurate and save time when writing the report.

5. Conclusion

The Smart Device Manager provides a simple solution to the problem described by Bright Minds Academy. The completed design uses a Python list of dictionaries to store device names, classroom locations and one of three required statuses: Online, Offline or Under Maintenance. Staff can add a device, update its status, view all records and search by device name through a clear menu. Validation reduces incorrect entries, while JSON storage keeps the records between sessions without requiring a database. The project also gave me experience of the full development process, from analysis and design to implementation, testing and evaluation. The solution is intentionally limited in scope, but its structure leaves room for later improvements such as editing, reporting, a graphical interface or database storage. Overall, the program meets the main requirements of the scenario while remaining simple enough for staff to understand and for the code to be maintained easily.

References

- [1] I. Sommerville, Software Engineering, 10th ed. Harlow, UK: Pearson, 2015.
- [2] Agile Business Consortium, “What is MoSCoW Prioritization?” [Online]. Available: <https://www.agilebusiness.org/resource/what-is-moscow-prioritization/>.
- [3] IBM, “What is the Software Development Lifecycle (SDLC)?” [Online]. Available: <https://www.ibm.com/think/topics/sdlc>.
- [4] Python Software Foundation, “5. Data Structures,” Python 3 Documentation. [Online]. Available: <https://docs.python.org/3/tutorial/datastructures.html>.
- [5] Python Software Foundation, “json — JSON encoder and decoder,” Python 3 Documentation. [Online]. Available: <https://docs.python.org/3/library/json.html>.
- [6] Python Software Foundation, “4. More Control Flow Tools,” Python 3 Documentation. [Online]. Available: <https://docs.python.org/3/tutorial/controlflow.html>.
- [7] Python Software Foundation, “8. Errors and Exceptions,” Python 3 Documentation. [Online]. Available: <https://docs.python.org/3/tutorial/errors.html>.
- [8] G. van Rossum, B. Warsaw and A. Coghlan, “PEP 8 — Style Guide for Python Code,” Python Enhancement Proposals. [Online]. Available: <https://peps.python.org/pep-0008/>.
- [9] IEEE Computer Society, “Software Engineering Body of Knowledge (SWEBOK) - Topics,” [Online]. Available: <https://www.computer.org/education/bodies-of-knowledge/software-engineering/topics>.

Appendix A - Short User Guide

1. Open main.py in PyCharm and run the file. The program loads any saved records from devices.json automatically.
2. Choose option 1 to add a device. Enter a unique device name and room. New devices are recorded as Offline by default.
3. Choose option 2 to update a device. Enter the device name and select 1 Online, 2 Offline or 3 Under Maintenance.
4. Choose option 3 to display all devices. The list shows Device Name, Room/Location and Status.
5. Choose option 4 to search by device name. The search is not case-sensitive.
6. Choose option 5 to save and exit. The program also saves immediately after adding a device or updating a status.