

SPC Assignment on Smart Home Device Manager System for Bright Minds Academy

ACCA5036 – UWTSD London

GITHUB – CODE ADDRESS URL

<https://github.com/CCVanusca/2411394.accca5036.git>

CRISTINEL-COSMIN VANUSCA | 2411394

Table of Content

1. Introduction.....	2
2. Analysis.....	3
3. Software Design And Development.....	4
4. Testing And Debugging Report.....	9
5. Evaluation.....	14
6. Conclusion.....	15
7. References.....	15
8. Python code.....	17

From Lecturer :

TASK DESCRIPTION

Project Scenario: Smart Home Device Manager

Bright Minds Academy, a small private tutoring centre known for its personalised learning approach, is piloting a smart classroom setup where various devices like smart lights, thermostats, and projectors are connected through a central system. Right now, the devices are being managed manually, and staff are finding it difficult to keep track of which device is connected, its status, or when it was last used. They want a simple system to keep track of smart devices, their locations, and connectivity status.

Task Overview:

For this project, you will create a simple program in Python to help Bright Minds Academy manage their smart home-style classroom devices. You don't need to use a database; you can use lists or dictionaries in Python to store the information.

Your program should let the staff do the following things:

1. **Add new devices:** The staff should be able to add new devices (like "Smart Light A1" or "Projector B2") and the room they are located in.
2. **Update device status:** The staff should be able to mark a device as online, offline, or under maintenance.
3. **View device list:** The staff should be able to see all devices, their locations, and current statuses.
4. **Simple Search:** The staff should be able to type in a device name and view its status and location.

Your program should be easy to use and understand. Write simple explanations of your code to show how it works. Focus on using lists or dictionaries to store the device information. You don't need to worry about real-time control or automation features for this project.

1. Introduction

Bright Minds Academy is a small private tutoring centre that is piloting a smart classroom environment. The scenario includes connected classroom devices such as smart lights, thermostats and projectors. At present, the devices are managed manually, which makes it difficult for staff to know which devices exist, where they are located and whether they are currently online, offline or under maintenance. The assessment specification therefore requires a simple software solution that can centralise this information and make it easier for staff to manage the classroom device inventory [1].

The proposed solution is a command-line Smart Home Device Manager developed in Python. The system is intentionally limited to the scope required by the scenario. It does not attempt to control physical devices or use a database. Instead, it stores device information in Python data structures while the program is running. This is appropriate for a small prototype because the assessment specifically permits lists or dictionaries instead of a database [1].

The main purpose of the program is to provide four core functions.

First, a member of staff can add a new device and record its room.

Second, the status of an existing device can be changed to Online, Offline or Maintenance.

Third, all stored devices can be displayed with their names, rooms and current statuses.

Fourth, a simple search can locate a device by name and display its information. Input validation is included to reduce incorrect entries.

The design follows a problem-decomposition approach. The overall problem is divided into smaller operations, each represented by a separate function. This makes the program easier to understand, test and maintain. Python is suitable for the prototype because it provides built-in lists, dictionaries, strings, loops, conditional statements and functions without requiring external software libraries [2].

The report documents

- the analysis,
- design decisions,
- implementation,
- testing and
- evaluation of the solution.

2. Analysis

Problem Definition

The central business problem is that Bright Minds Academy does not have a simple central method for recording classroom smart devices. Manual management can result in information being difficult to locate or becoming inconsistent. For example, a staff member may know that a projector exists but may not immediately know its assigned room or whether it is currently available. A small management program can provide a consistent structure for recording this information.

The proposed system treats each device as a record containing three primary attributes: device name, room and status. A device name identifies the device, the room identifies its physical classroom location and the status describes its current operational condition. This provides sufficient information to meet the core requirements without introducing unnecessary complexity.

Problem Decomposition

Problem decomposition is used to divide the overall problem into smaller tasks. The system is divided into the following operations:

1. Initialise the device collection.
2. Display a main menu.
3. Add a new device.
4. Validate the device name and room.
5. Update the status of an existing device.
6. Validate the selected status.
7. Display all devices.
8. Search for a device.
9. Return to the main menu until the user chooses to exit.

This decomposition separates user interaction from individual management tasks. Each major task is implemented as a function. This approach reduces duplication and makes individual functions easier to test.

Requirements Analysis

The requirements were derived from the assessment scenario. The system needs to be simple enough for staff without advanced programming knowledge to operate. It also needs to reject inappropriate values where validation is relevant [1].

The core functional requirements (FR) are:

- FR1 – Add Device: staff can create a new device record.
- FR2 – Record Location: staff can associate a room with the device.
- FR3 – Update Status: staff can change a device to Online, Offline or Maintenance.

FR4 – View Devices: staff can display all registered devices.

FR5 – Search Device: staff can enter a device name and retrieve its room and status.

FR6 – Input Validation: the program should reject empty fields, duplicate device names and invalid status selections.

FR7 – Menu Navigation: staff can repeatedly perform operations until they select Exit.

The non-functional (NFR) requirements are:

NFR1 – Usability: menu options and messages should be clear.

NFR2 – Reliability: valid operations should update and retrieve the correct record.

NFR3 – Maintainability: the code should be divided into logical functions with meaningful names and comments.

NFR4 – Performance: searches and updates should respond quickly for the small number of devices expected in a prototype.

NFR5 – Data consistency: device names and status values should follow consistent rules.

NFR6 – Security awareness: although this prototype does not implement authentication or network security, future versions should consider access control and protection of device information.

The last requirement is particularly relevant because real IoT (Internet of Things) environments have security concerns. NIST (The National Institute of Standards and Technology) identifies device identification, configuration, data protection, logical access, software updates and cybersecurity state awareness as important device cybersecurity capabilities [5]. The current assignment is much smaller and does not implement those controls, but recognising the limitation is useful when evaluating the prototype.

The requirements were also considered in terms of traceability. Each requirement should be visible in the design and later verified through testing. For example, FR3 requires a mechanism that identifies an existing device, presents valid status choices and changes the stored value. FR5 requires both a search input and a meaningful result when a match exists, together with an appropriate response when there is no match.

The requirements also influenced the decision not to implement a database. The scenario describes a small private tutoring centre and explicitly states that lists or dictionaries can be used. A database would be useful in a larger deployment, but it would introduce additional configuration, schema design and persistence concerns that are unnecessary for this prototype.

3. Software Design And Development

Development Approach

A simple iterative development approach was selected. The system can be developed in small stages: first the data structure, then the add function, status update, display, search, validation and finally testing. This is suitable for a small console program because each feature can be implemented and tested before the next feature is added. The assessment requires analysis, design, implementation, testing/debugging and documentation [1]. The development therefore follows the same sequence. During analysis, the requirements are identified. During design, the data model, functions, algorithm and user flow are defined. During implementation, these designs are translated into Python. Testing is then used to check both normal and invalid inputs.

System Architecture

The system uses a simple three-layer conceptual structure:

User interface layer:

The menu, prompts and output messages interact with the staff member.

Application logic layer:

Functions perform adding, updating, displaying and searching operations.

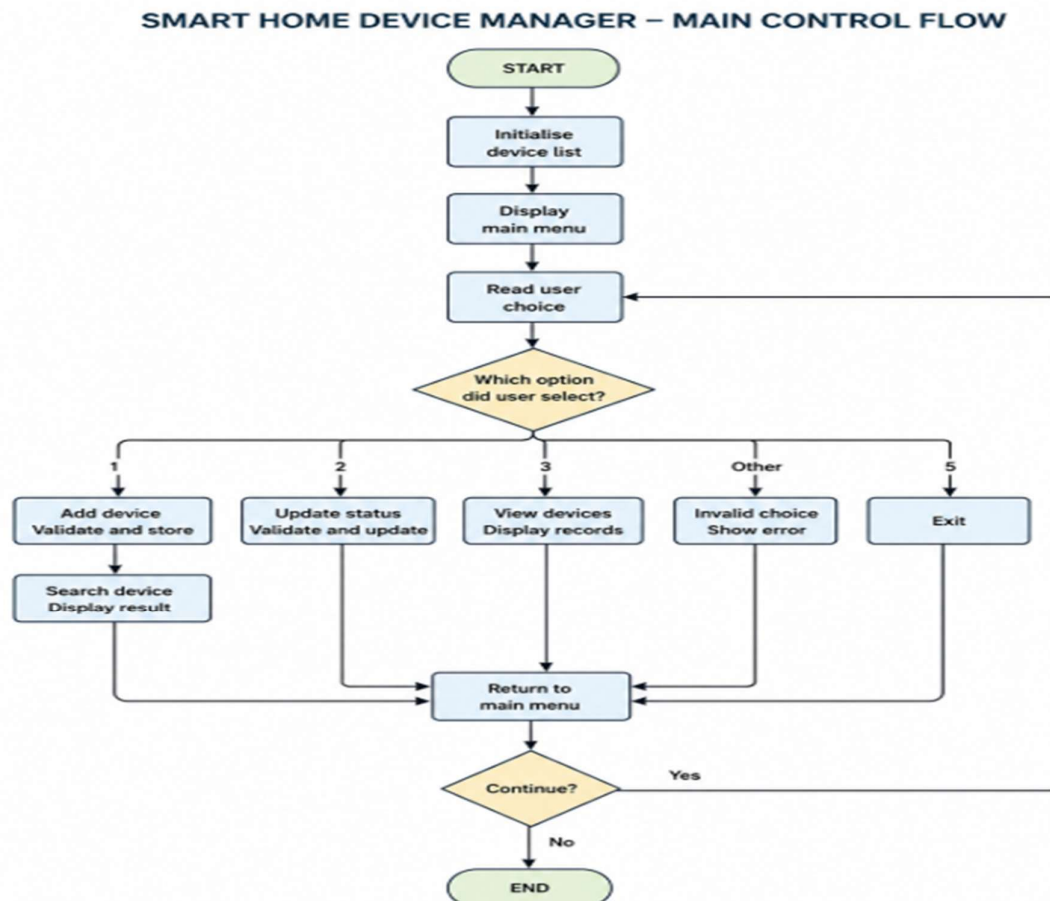
Data layer:

A Python list contains dictionaries representing the devices.

This is not a formal enterprise architecture, it is a simple separation of responsibilities. It makes the program easier to understand because each function has a clear role.

The design has a deliberately low level of coupling. The menu function is responsible for presenting options, while the feature functions perform the actual operations. The shared device collection allows an update made by one function to be visible when another function displays or searches the records. In a larger application, a separate data-access layer could reduce this coupling further, but that would be disproportionate to the scope of the assignment.

The interface follows a consistent interaction pattern. The user selects one operation,



enters the necessary information and receives either a success message or an explanation of the validation failure. The program then returns to the main menu. This reduces the risk that an input error terminates the application and supports repeated use during a staff session.

Flowchart

Pseudocode

Main program:

```
BEGIN
CREATE empty device list
ADD sample device records if required
for demonstration
REPEAT
DISPLAY menu
READ choice
IF choice = 1
CALL add device
ELSE IF choice = 2
CALL update status
ELSE IF choice = 3
CALL view devices
ELSE IF choice = 4
CALL search device
ELSE IF choice = 5
DISPLAY exit message
ELSE
DISPLAY invalid option message
END IF
UNTIL choice = 5
END
```

Add device:

```
BEGIN
READ device name
IF name is empty
DISPLAY validation message
RETURN
END IF
IF device name already exists
DISPLAY duplicate message
RETURN
END IF
READ room
IF room is empty
DISPLAY validation message
RETURN
END IF
SET status to Offline
CREATE device record
ADD record to device list
DISPLAY success message
END
```

Update status:

```
BEGIN
IF device list is empty
DISPLAY no devices message
RETURN
END IF
READ device name
SEARCH for matching device
IF device is not found
DISPLAY not found message
RETURN
END IF
DISPLAY valid status choices
READ new status
IF status is invalid
DISPLAY validation message
RETURN
END IF
UPDATE device status
DISPLAY success message
END
```

Search device:

```
BEGIN
READ device name
SEARCH each device in the list
IF a matching device is found
DISPLAY device name, room and status
ELSE
DISPLAY not found message
END IF
```

END

Data Structures

The main data structure is a list of dictionaries. A list is appropriate because the program needs to store a collection of devices and iterate over that collection. Each dictionary represents one device and stores related attributes using descriptive keys. Python documentation describes dictionaries as key-value structures where keys provide access to stored values [3].

A conceptual record is:

```
device = {  
    "name": "Smart Light A1",  
    "room": "Room 101",  
    "status": "Online"  
}
```

The complete collection is:

```
devices = [device1, device2, device3]
```

This structure was selected instead of three separate lists because the device's attributes remain grouped together. An alternative would be a list of lists, but dictionaries are more readable because the keys explain what each value represents. A class-based object model could also be used, but it would introduce additional complexity that is unnecessary for the small requirements of this assignment.

The main limitation is that the data exists only during execution. Closing the program loses the records. This is acceptable within the assessment scope because a database is not required, but persistent storage would be an appropriate future improvement.

The choice of a list of dictionaries also supports future extension. Additional keys, such as device type or last-used time, could be introduced without changing the fundamental collection model. Searching is implemented as a linear traversal of the list. With a small classroom inventory this is simple and fast enough. If the system grew to thousands of devices, a database index or dictionary keyed by a unique device ID would provide a more scalable lookup mechanism.

Variables

The principal variables are:

devices – the list containing all device dictionaries.

name – the device name entered by the user.

room – the classroom location.

status – the current state of the device.

choice – the menu option selected by the user.

new status – the status selected during an update.

device – a dictionary representing one record.

found device – the matching record returned by a search.

Meaningful variable names improve readability and reduce the likelihood of confusing different values.

Programming Constructs

Conditional statements are used to make decisions. The main menu uses if/elif/else to determine which operation corresponds to the user's choice. Conditional statements are also used for validation, for example when checking whether a device exists or whether a status is valid.

A while loop is used for the main menu because the application should continue running until Exit is selected. A for loop is used when searching through the device list and when displaying all records. Python provides control-flow statements such as conditional statements and loops as core language features [2].

Functions are used to decompose the program. The main functions are

display_menu(),
add_device(),
update_status(),
view_devices(),
search_device() and
main().

Functions reduce repetition and make testing more focused. For example, the search function can be tested separately from the menu.

String methods such as strip() and lower() can be used during validation. strip() removes unnecessary leading and trailing spaces, while lower() can support case-insensitive comparisons. These operations improve the user experience without requiring external libraries.

Input validation is important because user input cannot be assumed to be correct. The program checks for empty names, duplicate names, invalid menu choices and invalid status values. Where numeric conversion is required, Python's exception-handling mechanism is used to prevent a ValueError from terminating the application [4].

Python Modules and Libraries

The implementation does not require third-party libraries. This is an intentional design choice. The program can be implemented using Python's built-in language features, which reduces installation requirements and keeps the system easy to run.

Exception handling relies on Python's built-in exception mechanism rather than an external package. Python distinguishes syntax errors from exceptions that occur during execution, and try/except can be used to handle selected exceptions [4].

Implementation Decisions

The implementation prioritises simplicity, readability and direct mapping between requirements and functions. The device list is initialised at program startup. Each record contains only the attributes required for the prototype.

New devices are assigned an initial status of Offline. This prevents a newly created record from incorrectly appearing to be operational before a staff member has confirmed its state. The user can then change the status through the update function.

Duplicate device names are rejected because the search operation uses the name as the primary identifier. Allowing duplicates could make a search ambiguous. A more advanced system could use a unique device ID, but that is not necessary for this prototype.

The user interface is text-based. A graphical interface could be more visually accessible, but a command-line interface is sufficient for demonstrating the required programming concepts. It also keeps the implementation focused on Python control structures, data structures, functions and validation.

The program does not include real device communication. This is an intentional scope decision rather than a missing requirement. The assessment explicitly states that real-time control and automation features are not necessary [1].

Source Code

The complete source code is included in Appendix A and is also supplied as the separate Python file accompanying this report. The implementation contains six main functions and a main control loop. The code is organised using meaningful function names, comments/docstrings, consistent indentation and validation. It can be executed as a standard Python script.

4. Testing And Debugging Report

Importance of Testing

Testing is required to establish whether the software behaves as intended. A program can appear to work during a normal demonstration while still failing when users enter unexpected values. For this reason, the test strategy includes both valid and invalid inputs.

Testing is also connected to the requirements. Each functional requirement should have at least one test that demonstrates its behaviour. For example, the Add Device requirement is tested using a valid device and invalid input. The Search requirement is tested using both an existing and a non-existing device.

The ISTQB provides an established professional framework and terminology for software testing [6]. For this small project, the focus is on functional behaviour and input validation rather than a large-scale industrial testing process.

Types of Programming Errors

Syntax errors occur when Python code does not follow the language grammar. Examples include missing colons, incorrect indentation or unmatched brackets. Python identifies syntax errors before the affected code can execute [4].

Runtime errors, commonly represented as exceptions, occur during execution. For example, converting invalid text to an integer can produce `ValueError`. Attempting to access a missing dictionary key can produce `KeyError` [4].

Logical errors are different because the program runs but produces an incorrect result. For example, a search function could incorrectly report that a device is missing even when it exists. Logical errors are normally identified by comparing actual results against expected results.

Input-validation errors occur when the program accepts data that should have been rejected. For this project, examples include an empty device name, a duplicate device name or an invalid status. These errors are addressed by checking user input before modifying the data.

Testing Strategy

The testing strategy uses requirement-based functional tests and edge cases. The program is tested manually through the command-line interface. Each test records the input, expected result and actual result.

The main test groups are:

1. Menu validation.
2. Adding valid devices.
3. Rejecting invalid device names.
4. Rejecting duplicate devices.
5. Rejecting empty rooms.
6. Updating valid statuses.
7. Rejecting invalid statuses.
8. Displaying the device list.
9. Searching for an existing device.

10. Searching for a non-existing device.

11. Exiting the application.

The edge cases are particularly important because validation is explicitly required by the assignment [1].

The tests are designed around observable behaviour rather than the internal implementation. This means that a staff user can verify whether the program behaves correctly without needing to understand the Python source. Boundary and invalid-value tests are particularly useful because they demonstrate whether validation is actually enforced rather than merely described in the design. A successful test should provide the input, system response and comparison with the expected outcome.

Test Cases

Test Case
TC01 – Add valid device.
Input: Smart Light D4, Room 104. Expected result: A new device is stored with an initial status of Offline. Actual result: Device was added successfully with status Offline. Result: PASS.
TC02 – Empty device name.
Input: blank name. Expected result: The program rejects the value and asks for valid input. Actual result: “Error: device name cannot be empty.” Result: PASS.
TC03 – Duplicate device name.
Input: Smart Light A1. Expected result: The program rejects the duplicate. Actual result: “Error: a device with this name already exists.” Result: PASS.
TC04 – Update to Online.
Input: Smart Light D4 and Online. Expected result: Device status becomes Online. Actual result: “Smart Light D4 is now Online.” Result: PASS.
TC05 – Invalid status.
Input: status number 9. Expected result: Program rejects the value. Actual result: “Error: invalid status selection.” Result: PASS.
TC06 – View all devices.
Input: menu option 3. Expected result: all stored records are displayed with name, room and status. Actual result: the three initial devices were displayed correctly. Result: PASS.
TC07 – Search existing device.

Input: Smart Light D4. Expected result: matching room and status are displayed. Actual result: Room 104 and Online were displayed. Result: PASS.
TC08 – Search missing device.
Input: Unknown Device. Expected result: a not-found message is displayed. Actual result: “Device not found.” Result: PASS.
TC09 – Invalid menu option.
Input: 9. Expected result: an error message is displayed, and the main menu remains available. Actual result: “Error: invalid menu option. Please choose 1-5.” Result: PASS.
TC10 – Exit.
Input: 5. Expected result: the program terminates cleanly. Actual result: “Exiting Smart Home Device Manager. Goodbye.” Result: PASS.

The tests therefore cover all core operations and several validation cases. Representative evidence is included in Figures 2–5.

Fig.2 view all devices

```

--- Bright Minds Smart Device Manager ---
1. Add New Device
2. Update Device Status
3. View All Devices
4. Search Device
5. Exit
Select an option (1-5): 3

--- Registered Devices ---
Device: Samrt light A1 | Location: 01 | Status: Online
Device: Projector B2 | Location: 02 | Status: Offline
Device: Thermostat C1 | Location: 03 | Status: Maintenance

```

Figure 2 demonstrates the View All Devices function of the Smart Home Device Manager. After selecting option 3, the system successfully displays all registered devices together with their location and current status. The output shows three example devices: Smart Light A1, Projector B2 and Thermostat C1, with statuses of Online, Offline and Maintenance respectively. This confirms that the system can retrieve and present the stored device records clearly to the user.

Fig.3 Update device status and error

```
--- Bright Minds Smart Device Manager ---
1. Add New Device
2. Update Device Status
3. View All Devices
4. Search Device
5. Exit
Select an option (1-5): 2
Enter the name of the device to update: thermostat C1
Error: Device not found.

--- Bright Minds Smart Device Manager ---
1. Add New Device
2. Update Device Status
3. View All Devices
4. Search Device
5. Exit
Select an option (1-5): 2
Enter the name of the device to update: Thermostat C1
Select new status: 1. Online, 2. Offline, 3. Maintenance
Choice (1-3): 2
Success: Thermostat C1 status updated.
```

Figure 3 demonstrates the Update Device Status function and the system's input validation. In the first attempt, the first enters with a device name that is not found, and the system correctly displays an error message. The second attempt uses the correct device name, Thermostat C1, and successfully updates its status to Offline. This demonstrates that the program can handle invalid device searches and correctly update the status of an existing device.

Fig.4 Search device

```
--- Bright Minds Smart Device Manager ---
1. Add New Device
2. Update Device Status
3. View All Devices
4. Search Device
5. Exit
Select an option (1-5): 4
Enter the device name to search: Thermostat C1
Found: Thermostat C1 | Location: 03 | Status: Offline
```

Figure 4 demonstrates the Search Device function. The user selects option 4 and searches for Thermostat C1. The system successfully finds the device and displays its location (03) and current status (Offline). This confirms that the search function can retrieve and display the relevant information for an existing device.

Fig.5 Error of add, duplicate

```
--- Bright Minds Smart Device Manager ---  
1. Add New Device  
2. Update Device Status  
3. View All Devices  
4. Search Device  
5. Exit  
Select an option (1-5): 1  
Enter the name of the device (e.g., Smart Light A1): Thermostat C1  
Error: A device with this name already exists.
```

Figure 5 demonstrates the input validation implemented in the Add New Device function. The user attempts to add Thermostat C1, but the system detects that a device with the same name already exists and rejects the entry. This prevents duplicate device records and helps maintain data consistency within the device list.

Fig.6 Exit

```
--- Bright Minds Smart Device Manager ---  
1. Add New Device  
2. Update Device Status  
3. View All Devices  
4. Search Device  
5. Exit  
Select an option (1-5): 5  
Exiting system. Goodbye!
```

Figure 6 demonstrates the Exit function of the Smart Home Device Manager. The user selects option 5, and the system terminates the application successfully, displaying the confirmation message “Exiting system. Goodbye!” This confirms that the program can end the session cleanly when the user chooses the Exit option.

Debugging

Debugging is the process of locating and correcting defects. A structured approach was used: reproduce the problem, identify the relevant function, inspect the input and condition involved, make the correction, and repeat the test.

For example, if a search did not find a device because of differences in capitalisation, the comparison could be normalised using `lower()`. If a user entered an invalid menu value, validation could be added so the program returned to the menu rather than terminating. If a numeric conversion generated `ValueError`, `try/except` could be used where numeric input is required [4].

The distinction between testing and debugging is important. Testing reveals that the observed behaviour differs from the expected behaviour. Debugging investigates the cause and changes the implementation. A regression test is then performed to make sure the correction did not break another function.

5. Evaluation

Challenges Faced

One challenge was deciding how to represent each device. A simple list could store values, but using dictionaries provides clearer relationships between attributes. A class-based approach would provide more structure, but it would be unnecessary for a small procedural program. The list-of-dictionaries model therefore provides an appropriate balance between simplicity and readability.

Another challenge was input validation. A basic program can assume that users enter exactly what is expected, but this is unrealistic. Staff may enter blank values, different capitalisation or invalid menu choices. Validation was therefore incorporated into the main user interactions.

Search and update operations also require careful handling of devices that do not exist. Without a check, an update operation could fail or modify the wrong record. The design searches for a matching device and provides a clear message when no match is found.

A further challenge is the temporary nature of the data. Because the assignment does not require a database, the prototype stores records in memory. This makes the application simple, but it also means that information disappears when the program closes. This limitation was accepted because persistence is outside the required minimum scope.

Future Improvements

The first major improvement would be persistent storage. A JSON file or relational database could allow device information to remain available between program executions. Python's documentation includes support for saving structured data such as JSON [2]. A database would become more appropriate as the number of devices and users increased.

A second improvement would be a unique device identifier. Device names are understandable to staff but may not always be unique. An ID would provide a stronger key for updates and searches.

A graphical user interface could also improve usability. Buttons, forms, status indicators and search fields could make the system easier for non-technical users. Authentication could be added so that only authorised staff could change device information.

The system could also record last-used timestamps. This would directly extend the original scenario's desire to understand when devices were last used. A future version could additionally communicate with real IoT platforms and retrieve live status information.

Security should become more important as the system moves from a classroom prototype to a real networked system. NIST guidance identifies device identification, access control, data protection, software updates and cybersecurity state awareness as important IoT security capabilities [5]. A real deployment would therefore require authentication, access control, secure communications and appropriate network segmentation. These features are deliberately outside the present prototype.

A further improvement would be role-based access. Ordinary staff could be allowed to view and search devices, while authorised administrators could add devices or change maintenance states. Audit logging could also record who changed a status and when, providing accountability in a real deployment.

Self-Reflection

The project demonstrates that programming is not only about writing syntax. The most important part of the process was converting a general business problem into smaller technical requirements. Problem decomposition made it possible to design individual functions rather than attempting to build the whole program at once.

The project also strengthened understanding of data structures. A list provides a collection that can be iterated over, while dictionaries allow each device's attributes to be stored using descriptive keys. This combination is appropriate for a small management system and demonstrates how data representation affects the clarity of an implementation.

Testing provided another important learning outcome. A program that works with ideal input is not necessarily reliable. Invalid and unexpected values need to be considered during design, not only after implementation. The testing process therefore influenced the validation logic.

Finally, the project developed an understanding of maintainability. Separating functionality into functions, using meaningful names and keeping the program relatively simple makes future changes easier. The prototype could therefore be expanded without completely redesigning its core logic.

6. Conclusion

The Smart Home Device Manager provides a focused solution to the device-management problem described for Bright Minds Academy. The system uses Python data structures and functions to allow staff to add devices, record locations, update statuses, view device records and search for individual devices.

The design deliberately avoids unnecessary complexity. A list of dictionaries provides an understandable in-memory data model, while conditional statements, loops, functions and validation provide the required programming constructs. Testing covers both normal and invalid user inputs, supporting confidence that the main requirements have been implemented.

The prototype has limitations, particularly the absence of persistent storage, authentication and real device communication. However, these limitations are consistent with the scope of the assignment. Future development could introduce a database, graphical interface, unique device identifiers, usage timestamps and secure IoT integration. Overall, the project demonstrates the application of basic programming concepts to a realistic organisational problem while providing a foundation for future development.

7. References

- [1] University of Wales Trinity Saint David, “Level 5 Assessment Specification: Scripting and Programming Concepts, ACCA5036 – Coursework Programming Assignment with Associated Documentation,” 2026.
- [2] Python Software Foundation, “The Python Tutorial,” Python 3.14 Documentation, 2026. [Online]. Available: <https://docs.python.org/3/tutorial/>. [Accessed: Aug. 9, 2026].
- [3] Python Software Foundation, “Data Structures,” Python 3.15 Documentation, 2026. [Online]. Available: <https://docs.python.org/3/tutorial/datastructures.html>. [Accessed: Aug. 9, 2026].

- [4] Python Software Foundation, “Errors and Exceptions,” Python 3.14 Documentation, 2026. [Online]. Available: <https://docs.python.org/3/tutorial/errors.html>. [Accessed: Aug. 9, 2026].
- [5] M. Fagan, K. Megas, K. Scarfone, and M. Smith, “IoT Device Cybersecurity Capability Core Baseline,” NISTIR 8259A, National Institute of Standards and Technology, Gaithersburg, MD, USA, May 2020. doi: 10.6028/NIST.IR.8259A.
- [6] International Software Testing Qualifications Board, “ISTQB Glossary and Testing Resources,” ISTQB, 2026. [Online]. Available: <https://www.istqb.org/>. [Accessed: Aug. 9, 2026].

Python code :

Smart Home Device Manager System

Module: ACCA5036 - Scripting and Programming Concepts

```
def add_device(device_list):
    """Adds a new device to the dictionary with validation."""
    name = input("Enter the name of the device (e.g., Smart Light A1): ").strip()

    # Validation: Ensure name is not empty and doesn't already exist
    if not name:
        print("Error: Device name cannot be empty.")
    elif name in device_list:
        print("Error: A device with this name already exists.")
    else:
        location = input("Enter the room location: ").strip()
        print("Select initial status: 1. Online, 2. Offline, 3. Maintenance")
        status_choice = input("Choice (1-3): ")

        # Map numbers to status strings
        status_map = {"1": "Online", "2": "Offline", "3": "Maintenance"}
        status = status_map.get(status_choice, "Offline") # Default to Offline if invalid

        # Store in dictionary
        device_list[name] = {"room": location, "status": status}
        print(f"Success: {name} added to the system.")

def update_status(device_list):
    """Updates the status of an existing device."""
    name = input("Enter the name of the device to update: ").strip()
    if name in device_list:
        print("Select new status: 1. Online, 2. Offline, 3. Maintenance")
        choice = input("Choice (1-3): ")
        status_map = {"1": "Online", "2": "Offline", "3": "Maintenance"}
        device_list[name]["status"] = status_map.get(choice, "Offline")
        print(f"Success: {name} status updated.")
    else:
        print("Error: Device not found.")

def view_devices(device_list):
    """Displays all devices currently in the system."""
    if not device_list:
        print("\n--- No devices registered yet ---")
    else:
        print("\n--- Registered Devices ---")
        for name, info in device_list.items():
            print(f"Device: {name} | Location: {info['room']} | Status: {info['status']}")
```

```

def search_device(device_list):
    """Searches for a specific device by name."""
    name = input("Enter the device name to search: ").strip()
    if name in device_list:
        info = device_list[name]
        print(f"Found: {name} | Location: {info['room']} | Status: {info['status']}")
    else:
        print("Error: Device not found.")

def main():
    # Using a dictionary to store device information
    # Key = Device Name, Value = Dictionary of details
    devices = {}

    while True:
        print("\n--- Bright Minds Smart Device Manager ---")
        print("1. Add New Device")
        print("2. Update Device Status")
        print("3. View All Devices")
        print("4. Search Device")
        print("5. Exit")

        choice = input("Select an option (1-5): ")

        if choice == '1':
            add_device(devices)
        elif choice == '2':
            update_status(devices)
        elif choice == '3':
            view_devices(devices)
        elif choice == '4':
            search_device(devices)
        elif choice == '5':
            print("Exiting system. Goodbye!")
            break
        else:
            print("Invalid choice. Please enter a number between 1 and 5.")

if __name__ == "__main__":
    main()

```