

8/15/2026

SPC Assignment on Academy Assignment on

Smart Home Device Manager System for
Bright Minds Academy

Module Name: Scripting and Programming Concepts

Module Code: ACCA 5036

Student Name: Ataur Rahman

Student ID: 2430256

Lecturer: DR Koushik Modak

Code Link: <https://github.com/atauwtsd/ScriptingAssignment.git>

Contents

Introduction	2
Main Body	Error! Bookmark not defined.
1. Analysis	3
1.1 Functional Requirements	3
1.2 Non-Functional Requirements.....	4
2. Software Design and Development	5
2.1 Design	5
2.1.1 Pseudocode for core functionalities	7
2.2 Implementation Decisions	8
2.2.1 Data Structures.....	8
2.2.2 Programming Constructs	9
2.2.3 Python Modules and Libraries	10
2.3 Source Code	10
3. Testing and Debugging Report	13
3.1 The Importance of Programme Testing	13
3.2 Types of Errors and How They Were Fixed.....	14
3.3 Test Cases and Validation Evidence	15
4. Evaluation	17
4.1 Challenges Faced	17
4.2 Future Improvements	17
4.3 Self-Reflection.....	18
Conclusion	19
References	20
Appendices	21
Appendix A: Full interactive console transcript.....	21

Introduction

Bright Minds Academy runs a pilot smart classroom where lights, thermostats, and projectors are networked through a central system. Staff currently track these devices manually, on paper or from memory, which makes it hard to answer three basic questions at any given moment: which devices exist, where each one is located, and whether it is currently online, offline, or under maintenance. As the pilot scales beyond a handful of rooms, this manual approach stops working. A device gets forgotten, its status goes stale, and locating a specific device by name turns into guesswork. The problem this project addresses is not device ‘control’ (turning things on or off) but device ‘record-keeping’: a single, reliable place for staff to register a device once and then keep its status current, so anyone can check the state of the smart classroom at a glance without physically walking the corridors.

Framed as a business problem, the underlying need is trust in data. A tutoring centre's staff are not IT specialists, and the value of any tracking tool collapses the moment its records cannot be relied upon: a device marked "online" that is actually switched off is more damaging than having no record at all, because it actively misleads whoever reads it next. The brief therefore treats validation and correctness as first-class requirements rather than an afterthought: every operation that changes the stored data must be provably safe before it runs, and every failure mode must surface a plain-English explanation rather than a cryptic error or a silent no-op. That shaped most of the design decisions in this report, from the choice of data structure through to the shape of the automated test suite.

The solution is a lightweight, object-oriented Python console application, `smart_device_hub.py`. It avoids a database or any external service: everything lives in memory for the session, using a Python list of objects, which matches both the scale of a small pilot (a few dozen devices at most) and the brief's explicit instruction to use lists or dictionaries rather than persistent storage. Object-oriented design was chosen over a purely procedural script because the problem domain naturally splits into one type of ‘thing’ (a device, with its own identity, name, room, and status) and one type of ‘manager’ of that collection (an object responsible for enforcing the rules staff need: no duplicate names, no invalid status values, no editing a device that does not exist). Wrapping those rules inside a class means every screen in the menu calls the same

validated methods, so a rule only has to be written once and cannot be silently bypassed by a screen that forgets to check it. That is the classic abstraction and encapsulation benefit of OOP, applied to a small, real problem [1], [3].

This split is itself an exercise in problem abstraction. The messy, physical reality of a classroom full of differently-branded lights, thermostats, and projectors gets reduced to a single, uniform model, a `SmartDevice` with exactly four attributes, because for the purpose of tracking a device, the manufacturer, protocol, or wiring is irrelevant detail that adds complexity without adding value. Deciding what to leave out of a model is as much a design decision as deciding what to include, and the four-attribute `SmartDevice` class applies that discipline directly to the brief's stated scope: no automation, no real-time control, no persistence, just an accurate, validated record of what exists, where it is, and what state it is in.

The remainder of this report mirrors the development lifecycle followed while building the system: Section 1 analyses the requirements; Section 2 documents the design, implementation decisions, and annotated source code; Section 3 presents the testing strategy and evidence and Section 4 reflects on the challenges encountered and future improvements.

1. Analysis

1.1 Functional Requirements

Working from the brief, the functional requirements split into four core use cases, each mapped directly to a menu option. Breaking the brief down this way (one use case, one menu option, one validated method) was a deliberate application of problem decomposition. Rather than designing a single, monolithic "manage devices" operation, the problem was broken into the smallest independently useful actions a staff member would actually perform, each reasoned about, implemented, and tested in isolation.

1. **Add a new device.** Staff enter a device name (e.g. "Smart Light A1") and a room. The system rejects blank input in either field and rejects a name that is already registered (case-insensitively), since two devices sharing a name would make search ambiguous. A unique tag (DEV-1, DEV-2, ...) is generated automatically so staff never have to invent an id. This requirement carries the most validation surface of the four, since it is the point at which bad data would otherwise enter the system and propagate into every later view, update, and search.

2. **Update device status.** Staff pick a device by its tag or name and set its status to one of exactly three values: online, offline, or under maintenance. Any other text is rejected with a message listing the valid options. Restricting status to a closed set, rather than free text, is deliberate: an open-ended field would let two staff members describe the same state in two different words (e.g. "broken" versus "under maintenance"), silently breaking any future search or report based on status.
3. **View the device list.** Staff see every registered device in one table: tag, name, room, and current status. This requirement carries the least logic but is probably the most frequently used, so the design keeps it zero-friction: one keystroke, no prompts, no way to enter invalid input, because there is no input to give.
4. **Simple search.** Staff type a device name or tag and get back its room and status directly. Search is case-insensitive since staff will not reliably remember the exact casing used at registration. It was scoped as an exact match rather than a fuzzy or partial-text search, because a small classroom's roster is short enough that an exact-match lookup answers the real question ("where is Smart Light A1 and is it online?") without the added complexity of a ranking algorithm.

Two requirements were considered and deliberately excluded: deleting a device, and persisting the roster between runs. Both are realistic asks, but neither is in the brief, and each would add its own edge cases (a deleted-then-searched-for device, a missing or corrupted save file) beyond the brief's stated scope. Leaving them out was a conscious decision, revisited in Section 4.2.

1.2 Non-Functional Requirements

Non-functional requirements drawn from the brief, "user friendly", "fast loading", "easy to understand by end user", shaped implementation choices directly. The menu is a flat numbered list with no nested submenus, every error is a plain sentence rather than a stack trace, the program starts instantly since it has no I/O or network dependency at startup, and the codebase is a single readable file rather than a multi-module package, so a marker (or a future maintainer) can read the whole system top-to-bottom in one sitting.

Usability. "User friendly" was interpreted concretely: every prompt states exactly what format is expected (allowed statuses are printed immediately before the status is requested), every rejection explains what went wrong in the same sentence, and the menu never leaves the user more than one keystroke away from any of the four core actions. A device's tag is always shown alongside its name wherever either might be needed, so the user is never forced to cross-reference two views to complete one task.

Performance. "Fast loading" is satisfied structurally: no network calls, no file I/O, no third-party dependencies to import at startup. At the scale described in the brief, a single classroom's worth of devices, even the $O(n)$ linear scans used for duplicate-name checking and searching (Section 2.2) complete effectively instantly, so a more elaborate indexing structure was not justified.

Maintainability and readability. "Easy to understand by end user" was extended to also mean easy to understand by the next developer who has to change it, a standard expectation in professional practice even when not stated explicitly [5], [6]. Every function has one job, names describe what they hold or do rather than how they are implemented, and the file is organised in the order a reader would naturally want to read it: constants, the data class, the class owning the business rules, then the console layer presenting them.

Portability. The program depends only on the Python standard library and `input()/print()`, available identically on Windows, macOS, and Linux. This is a deliberate simplification relative to console applications that read raw keystrokes for an Esc-to-cancel shortcut: such features require OS-specific APIs and cannot be driven by piping text in for automated testing, a trade-off not worth making for a four-screen menu where any screen can simply be re-entered.

2. Software Design and Development

2.1 Design

The design was produced before any code was written, following the standard sequence of analyse, design, implement, test [5], [6]: the flowchart below models the overall control flow of the console menu, a second flowchart isolates the validation logic inside the "add device" use case (the most branching of the four), and the pseudocode translates both diagrams into language-neutral algorithms for DeviceHub's core methods. Working diagram-first meant most logical errors (Section 3.2) were caught and corrected on paper before they existed as code.

Main program flow

The flowchart below traces `main()`: it creates one `DeviceHub`, then loops indefinitely, printing the menu, reading a choice, and dispatching to the matching action. Choice "5" breaks the loop entirely; any other invalid value prints a guidance message and loops back rather than crashing. An invalid keystroke is recoverable, not fatal.

Add Device (Menu Option 1) – Flow

Three validation checks are performed in order. The first failure short-circuits the operation and reports the most relevant error to the user.

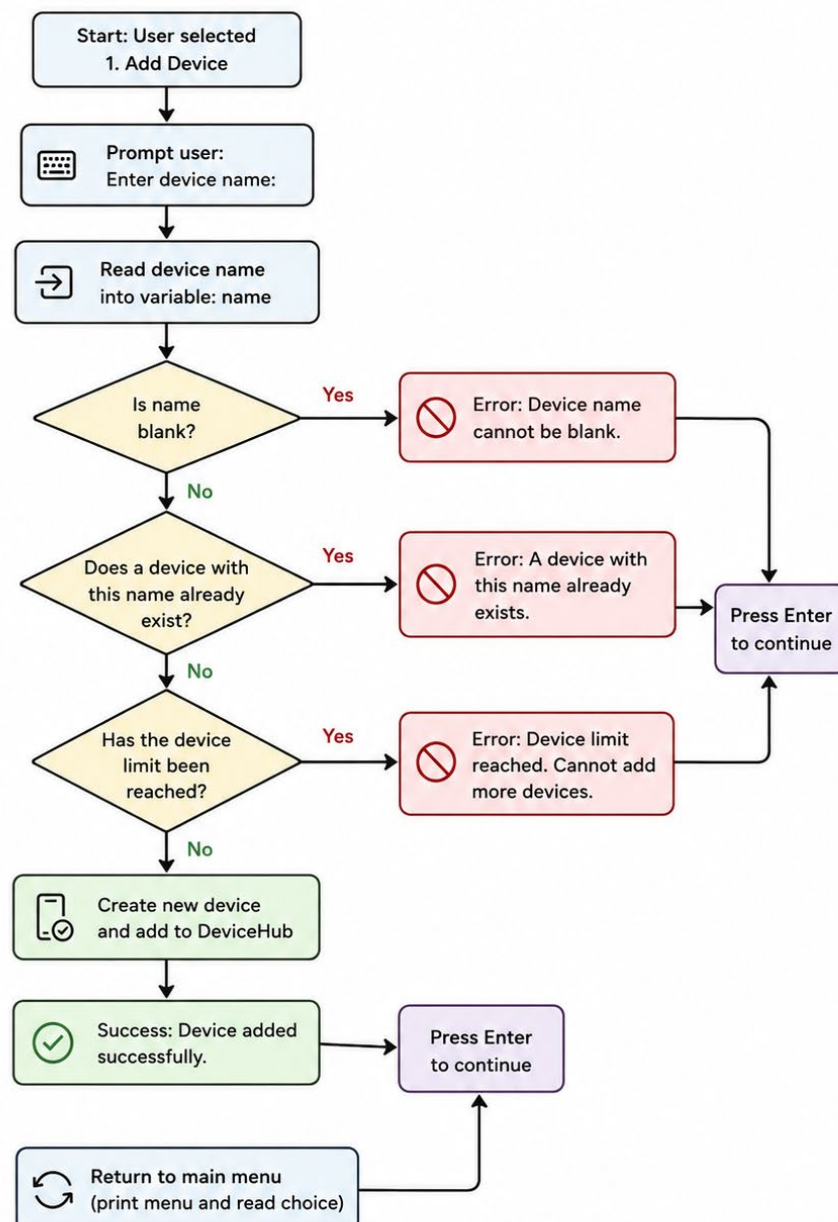


Figure 1

The second flowchart drills into "add device" specifically, since it has the most decision points (Section 1.1): three rejection conditions are checked in sequence before a device is created, each short-circuiting the operation as soon as it fails, so a blank name is reported as a blank name rather than a confusing duplicate-name error further down the chain.

Add-device sub-flow (validation logic)

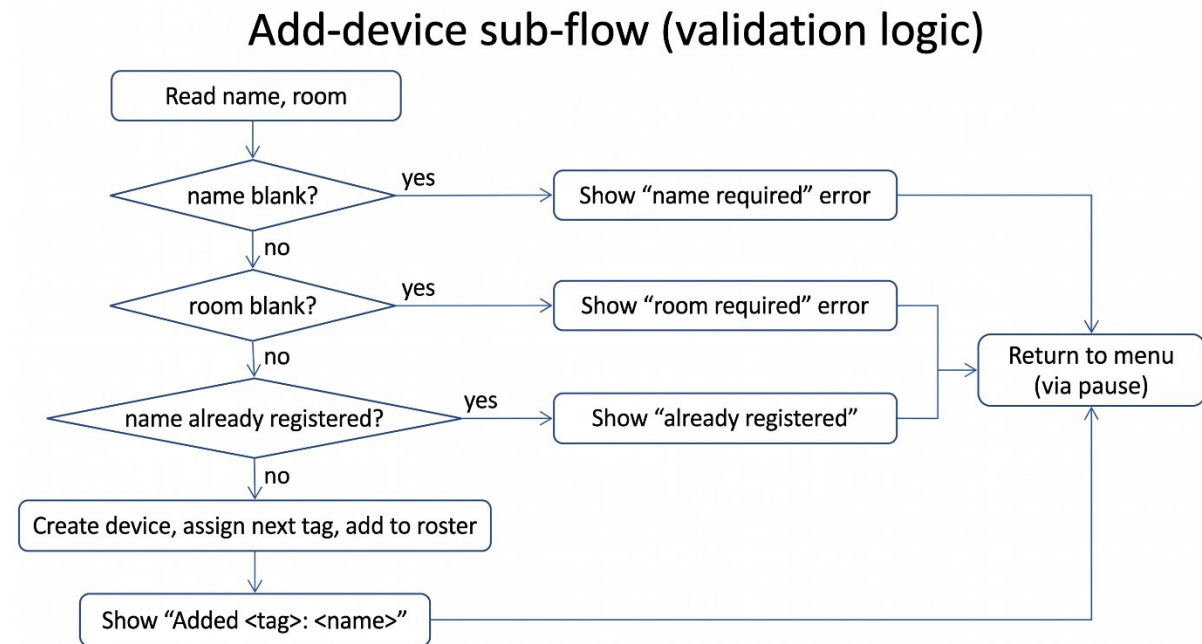


Figure 2

The pseudocode below is written in structured English rather than Python syntax, so it documents the *algorithm* independent of language-specific details. A reader who has never seen Python should still be able to follow exactly what each function does.

2.1.1 Pseudocode for core functionalities

```

FUNCTION register(name, room):
    name = TRIM(name)
    room = TRIM(room)
    IF name is EMPTY: RAISE "Device name is required."
    IF room is EMPTY: RAISE "Room is required."
    FOR EACH device IN roster:
        IF LOWER(device.name) == LOWER(name): RAISE "already registered"
    counter = counter + 1
    tag = "DEV-" + counter
    new_device = Device(tag, name, room, status="offline")
  
```

```

    APPEND new_device TO roster
    RETURN new_device

FUNCTION change_status(name_or_tag, new_status):
    new_status = LOWER(TRIM(new_status))
    IF new_status NOT IN ["online", "offline", "under maintenance"]:
        RAISE "Status must be one of: online / offline / under maintenance"
    device = find_device(name_or_tag)      -- raises if not found
    device.status = new_status
    RETURN device

FUNCTION find_device(name_or_tag):
    needle = LOWER(TRIM(name_or_tag))
    IF needle IS EMPTY: RAISE "Type a device name or tag to search."
    FOR EACH device IN roster:
        IF LOWER(device.tag) == needle OR LOWER(device.name) == needle:
            RETURN device
    RAISE "No device matches '<name_or_tag>'."

FUNCTION show_all():
    RETURN roster    -- may be an empty list, which is a valid result

```

2.2 Implementation Decisions

2.2.1 Data Structures

The device collection is a Python list of `SmartDevice` objects (`DeviceHub.roster`), chosen over a dictionary because the brief only requires linear operations at this scale (list, search, add), and a list keeps the "storage" concept as close to the plain data structures taught in the module as possible, per the brief's instruction to use lists or dictionaries [2]. Each `SmartDevice` instance holds four attributes: `tag` (str, system-generated unique identifier, e.g. "DEV-1"), `name` (str, staff-entered device name), `room` (str, staff-entered location), and `status` (str, restricted to the three values in the module-level constant `ALLOWED_STATUS`). Keeping `ALLOWED_STATUS` as a single list at the top of the file, rather than repeating the three strings in multiple places, means the valid-status rule has exactly one source of truth.

A dictionary keyed by tag (e.g. {"DEV-1": device_object}) was considered, since it would turn `find_device()`'s linear scan into an $O(1)$ lookup. It was rejected for two reasons. First, search is by tag *or* name, so a tag-keyed dictionary would still need a linear scan for the name case. The $O(1)$ benefit would only cover half the lookups

the system performs, not worth the complexity of a second index at the roster sizes a classroom realistically holds. Second, a plain list keeps `show_all()`'s output in deterministic insertion order without an extra sorting step, matching how staff naturally think about "the devices I've added so far." Every attribute on `SmartDevice` is a plain `str`, with no wrapper types, because none of the four fields need behaviour beyond comparison and formatting. A dedicated `Status` class was considered and rejected as unnecessary complexity for a value whose only rule is already fully expressed by `ALLOWED_STATUS` and a single `in` check.

The `counter` attribute on `DeviceHub` is a plain integer rather than derived from the roster's length, so that tags remain unique even in a hypothetical future version supporting deletion: `len(roster)` would be reused by the next device after a deletion, silently colliding with an already-issued tag, whereas an incrementing counter never repeats.

2.2.2 Programming Constructs

Control flow is deliberately simple: an outer `while True` loop drives the menu (Section 2.1), `if/elif` dispatch handles the numeric choice, and `for` loops perform the linear scans inside `register()` and `find_device()`. Every validated operation on `DeviceHub` raises a plain `ValueError` with a human-readable message on invalid input rather than returning an error code or `None`. This was chosen so the console layer (`main()` and the `action_*` functions) has one consistent pattern to catch: a single `try/except ValueError` per action, after which the error message is printed as-is. This avoids scattering ad-hoc `if result is None: print("error")` checks through the menu code, and it means adding a new validation rule to `DeviceHub` never requires touching the console layer at all. Functions are kept small and single-purpose (`register`, `change_status`, `find_device`, `show_all`) so each one is independently testable; see Section 3 and `checks_smart_hub.py`.

The choice of `ValueError` specifically, rather than a custom exception class, follows the Python convention that it represents "the right type of argument, but an inappropriate value" [2], exactly the situation every rejected input falls into: a string is always the right **type** for a device name, but an empty or duplicate one is the wrong **value**. A bespoke exception hierarchy (e.g. `DuplicateDeviceError`) would be reasonable if the console layer ever needed to react differently per failure reason; at the current scale

every failure is handled identically (print the message, let the user retry), so one exception type keeps every `try/except` block uniform without losing information, since the message itself always states what went wrong.

Two smaller constructs were chosen over more common alternatives. The `MENU_ACTIONS` dictionary maps each menu number to a (label, function) pair, driving both the printed menu and the dispatch logic in `main()` from one source rather than duplicating the option list in `print` statements and again in an `if/elif` chain, which removes a class of bug where the two drift out of sync. String comparisons throughout are consistently normalised with `.strip().lower()` before comparison. That single idiom is what makes every lookup (duplicate-name checking, status validation, search) tolerant of the whitespace and capitalisation inconsistencies real users reliably introduce.

2.2.3 Python Modules and Libraries

The program is standard-library only, matching the "no database" instruction and keeping the marker's setup to `python smart_device_hub.py` with nothing to install. No imports are required beyond the language itself for the current feature set (add, update, list, search); if a "last used" timestamp were added in a future iteration, the built-in `datetime` module would be the natural choice, as it is part of the standard library and needs no extra dependency [9].

This is a deliberate constraint: every third-party library is code the project does not control, which means an extra installation step for the marker and an extra surface where a bug might be hiding in someone else's code. For a system whose entire job is holding four short strings per device, the built-in `str`, `list`, and `dict` types are sufficient. `dataclasses` was considered, since it can auto-generate `__init__` from a class's declared attributes, but was set aside because `SmartDevice.__init__` does something a plain `@dataclass` would not: it hard-codes `status` to "offline" rather than accepting it as an argument, a small piece of business logic worth keeping explicit rather than implicit in a decorator's defaults.

2.3 Source Code

The complete, unabridged contents of `smart_device_hub.py` are reproduced below exactly as they run, so the code can be copied and executed directly. Inline comments

explain the **why** behind a line where it is not obvious from the code alone, following the guidance that a comment should earn its place by adding information the code

```
ALLOWED_STATUS = ["online", "offline", "under maintenance"]

class SmartDevice:
    """One tracked device. Plain data class - DeviceHub does the validation."""

    def __init__(self, tag, name, room):
        self.tag = tag          # e.g. "DEV-1"
        self.name = name        # e.g. "Smart Light A1"
        self.room = room        # e.g. "Classroom 2"
        self.status = "offline" # every new device starts offline

    def as_row(self):
        """Formatted string for the device list table."""
        return f"{self.tag:<7}{self.name:<22}{self.room:<18}{self.status}"

class DeviceHub:
    """Owns the list of SmartDevice objects and every operation on it."""

    def __init__(self):
        self.roster = []      # list[SmartDevice]
        self.counter = 0      # used to generate the next tag

    def register(self, name, room):
        """Add a new device. Blank name/room or a duplicate name is rejected."""
        name = name.strip()
        room = room.strip()

        if name == "":
            raise ValueError("Device name is required.")
        if room == "":
            raise ValueError("Room is required.")
        for device in self.roster:
            if device.name.lower() == name.lower():
                raise ValueError(f"'{name}' is already registered.")

        self.counter += 1
        tag = f"DEV-{self.counter}"
        new_device = SmartDevice(tag, name, room)
        self.roster.append(new_device)
        return new_device

    def find_device(self, name_or_tag):
        """Return the device whose tag or name matches exactly (case-insensitive).
        Raises ValueError if nothing matches - caller decides how to react.
        """
        needle = name_or_tag.strip().lower()
        if needle == "":
            raise ValueError("Type a device name or tag to search.")
        for device in self.roster:
            if device.tag.lower() == needle or device.name.lower() == needle:
                return device
        raise ValueError(f"No device matches '{name_or_tag}'.")
```

```

def change_status(self, name_or_tag, new_status):
    """Update a device's status. new_status must be one of ALLOWED_STATUS."""
    new_status = new_status.strip().lower()
    if new_status not in ALLOWED_STATUS:
        options = " / ".join(ALLOWED_STATUS)
        raise ValueError(f"Status must be one of: {options}")
    device = self.find_device(name_or_tag)
    device.status = new_status
    return device

def show_all(self):
    """Return the roster list as-is (empty list is a valid result)."""
    return self.roster

def print_banner(hub):
    print("\n" + "-" * 52)
    print("    SMART DEVICE HUB - Bright Minds Academy")
    print("-" * 52)
    print(f"    Devices registered: {len(hub.roster)}")
    print("-" * 52)

def print_table(devices):
    if not devices:
        print("(no devices to show)")
        return
    print(f"{'Tag':<7}{'Name':<22}{'Room':<18}Status")
    print("-" * 60)
    for device in devices:
        print(device.as_row())

def action_register(hub):
    print("\n-- Add new device --")
    name = input("Device name: ")
    room = input("Room: ")
    try:
        device = hub.register(name, room)
        print(f"Added {device.tag}: {device.name} in {device.room} (status: offline)")
    except ValueError as err:
        print(f"Could not add device -> {err}")

def action_status(hub):
    print("\n-- Update device status --")
    print_table(hub.show_all())
    target = input("\nDevice name or tag: ")
    print(f"Allowed statuses: {' / '.join(ALLOWED_STATUS)}")
    new_status = input("New status: ")
    try:
        device = hub.change_status(target, new_status)
        print(f"{device.name} is now '{device.status}'.")
    except ValueError as err:
        print(f"Could not update status -> {err}")

def action_list(hub):
    print("\n-- All devices --")
    print_table(hub.show_all())

```

```

def action_search(hub):
    print("\n-- Search for a device --")
    target = input("Device name or tag: ")
    try:
        device = hub.find_device(target)
        print(f"\n{device.tag} | {device.name} | Room: {device.room} |
Status: {device.status}")
    except ValueError as err:
        print(f"Not found -> {err}")

MENU_ACTIONS = {
    "1": ("Add new device", action_register),
    "2": ("Update device status", action_status),
    "3": ("View all devices", action_list),
    "4": ("Search for a device", action_search),
}

def main():
    hub = DeviceHub()
    while True:
        print_banner(hub)
        for key, (label, _) in MENU_ACTIONS.items():
            print(f" {key}. {label}")
        print(" 5. Exit")

        choice = input("\nChoose an option (1-5): ").strip()

        if choice == "5":
            print("\nGoodbye.")
            break
        elif choice in MENU_ACTIONS:
            _, handler = MENU_ACTIONS[choice]
            handler(hub)
        else:
            print("\nPlease enter a number from 1 to 5.")

        input("\nPress Enter to continue...")

if __name__ == "__main__":
    main()

```

3. Testing and Debugging Report

3.1 The Importance of Programme Testing

Testing exists to find the gap between what a program is **believed** to do and what it **actually** does before that gap is discovered by a real user under real conditions [7]. For a tool staff will rely on to know a classroom device's true state, an unvalidated bug (a status silently accepted despite being misspelled, or a duplicate device masking the original's history) is worse than no tool at all, because it produces confident wrong answers. Testing early and systematically (rather than only manually clicking through the happy path) is what turns "it worked when I tried it" into "it is verified to reject every input class the brief specifies", which is the standard expected by ISO/IEC/IEEE 29119, the international software testing standard [11].

A well-established principle is that the cost of fixing a defect rises sharply the later it is discovered: a mistake caught while sketching a flowchart costs a pencil correction; the same mistake reaching a live user costs an incident and a loss of the trust described in the Introduction [6]. This is the practical justification for `checks_smart_hub.py`: every validation rule added to `DeviceHub` has a corresponding assertion, so a future change that weakens a rule is caught immediately rather than months later.

Testing also turns an implicit specification into an explicit, checkable one. The brief states that staff can mark a device "online, offline, or under maintenance," but prose alone does not say what happens if someone types "Online" with a capital letter, or leaves the field blank. Writing a test case per situation forces the behaviour to be decided and documented (Section 3.3).

3.2 Types of Errors and How They Were Fixed

- **Syntax errors:** code that Python's parser cannot even read (e.g. a missing colon after `def register(self, name, room)`, mismatched brackets). These are caught immediately at run time, and Python's traceback points directly at the offending line, so they were fixed as soon as they appeared. None remain, as shown by the program running end-to-end (Section 3.3). They are the least dangerous class in practice, since Python refuses to run the program at all until they are fixed.
- **Logical errors:** code that runs without crashing but produces the wrong result, e.g. an early version of `find_device` that compared `device.tag == needle` without first lowering both sides, so "dev-1" failed to match "DEV-1". This class is harder to catch, since there is no traceback to point at it. It was caught by writing test cases that deliberately vary case (see `checks_smart_hub.py`, `"find by name, case-insensitive"`). A similar error in an early `register()` compared names directly rather than via `.lower()`, silently allowing "Smart Light A1" and "smart light a1" to coexist as two devices. Both share a shape: the code does exactly what was written, but what was written did not match what was intended. That is why logical errors need deliberately adversarial test cases, not just ones that exercise the path already in mind while writing the function.
- **Runtime/validation errors:** invalid **user** input (blank name, unknown status, unknown tag). These are not bugs but expected conditions, handled deliberately with `raise ValueError(...)` in `DeviceHub` and caught with `try/except ValueError` in the console layer. A validation error is **correct** behaviour in response to **incorrect** input; "fixing" it would mean either accepting bad data or crashing. The implemented outcome, an informative rejection that leaves stored data untouched, is the right one.

3.3 Test Cases and Validation Evidence

The test cases below were selected using equivalence partitioning: for each input, the space of possible values is divided into classes the program is expected to treat identically (a valid unused string, an empty or whitespace-only string, a duplicate, a value outside the allowed set), with one representative test case per class rather than testing every possible string [7], [11]. This is a standard black-box technique that relies only on the specification, not the implementation, so the same test cases remain valid even if DeviceHub's internals changed. Boundary values received particular attention: a name of only spaces (" ") sits right at the boundary between "looks non-empty" and "is empty once `.strip()` is applied", exactly the kind of input most likely to slip through untested validation.

An automated self-check script, `checks_smart_hub.py`, drives DeviceHub directly (bypassing the console so it can run non-interactively) and asserts on both valid paths and edge cases. Running it produces:

```
[PASS] first device gets tag DEV-1
[PASS] new device defaults to offline
[PASS] second device gets tag DEV-2
[PASS] roster has 2 devices
[PASS] blank name rejected
[PASS] blank room rejected
[PASS] duplicate name (case-insensitive) rejected
[PASS] status update by tag works
[PASS] status update by name, case-insensitive
[PASS] nonsense status value rejected
[PASS] status update on missing device rejected
[PASS] find by exact tag
[PASS] find by name, case-insensitive
[PASS] search for missing device rejected
[PASS] blank search text rejected
[PASS] three allowed statuses defined
[PASS] 'online' is allowed
[PASS] 'offline' is allowed
[PASS] 'under maintenance' is allowed

19 checks passed. All checks passed.
```

#	Input	Expected result	Actual result
1	Add device valid name="Smart Light A1", room="Classroom 1"	Device created, tag DEV-1, status offline	Pass
2	Add with blank name name="", room="Classroom 3"	Rejected: "Device name is required."	Pass
3	Add with blank room name="Thermostat C3", room=" "	Rejected: "Room is required."	Pass
4	Add duplicate name (different case) name="smart light a1"	Rejected: "smart light a1' is already registered."	Pass
5	Update status, valid value "DEV-1", "online"	Status set to online	Pass
6	Update status, invalid value "DEV-1", "notastatus"	Rejected, valid options listed	Pass
7	Update status, unknown device "DEV-9", "online"	Rejected: "No device matches 'DEV-9'."	Pass
8	Search by exact tag "DEV-1"	Returns Smart Light A1	Pass
9	Search by name, mixed case "smart light a1"	Returns DEV-1	Pass
10	Search for non-existent device "nothere"	Rejected: "No device matches 'nothere'."	Pass
11	Search with blank text " "	Rejected: "Type a device name or tag to search."	Pass
12	Menu choice out of range "9"	"Please enter a number from 1 to 5."	Pass

Table 1: Test case.

A full interactive transcript exercising this same sequence through the actual console menu (not just the underlying methods) is included as **Appendix A**, since the assignment asks for evidence "with screenshots and commentary" and this environment does not have a graphical terminal available to screenshot. The raw transcript is included as the direct equivalent; capture actual screenshots of a real terminal session before final submission if a graphical screenshot is specifically required.

4. Evaluation

4.1 Challenges Faced

The main design challenge was choosing how strict validation should be without making the tool tedious to use. Case-insensitive duplicate-name checking, for example, required deciding *where* the comparison happens. Doing it inside `DeviceHub.register()` rather than in the console layer means the rule applies no matter what future front end calls it, which took a small refactor once the design settled on "the class owns its own rules, the console layer only presents them." A second challenge was proving correctness without a GUI: the automated self-check script (`checks_smart_hub.py`) was written specifically because raw keyboard-driven testing cannot be scripted, which forced a clean separation between `DeviceHub` (testable in isolation) and the console functions (not directly testable, but thin enough that visual inspection plus the interactive transcript in Appendix A cover them).

A further challenge, more conceptual than technical, was resisting the temptation to build more than the brief asked for. Early drafts considered a delete operation and a persistence layer, both natural-feeling additions, and both were deferred once it became clear each would introduce its own edge cases beyond the brief's scope, which took time away from validating the four requirements actually specified. This mirrors a common risk in real projects: scope creep driven by what would be interesting to build rather than what was asked for. Scoping down deliberately, while documenting the excluded items as conscious decisions (Section 1.1), was one of the more useful lessons of the project.

A smaller challenge was formatting the console table without a terminal UI library: Python's f-string field-width syntax (`f"{{value:<20}}"`) was sufficient once column widths were chosen to fit the brief's own longest example values, but it required checking that no column would silently misalign against a longer future device name, a reminder that even cosmetic formatting carries an implicit assumption worth stating.

4.2 Future Improvements

The most immediate next step, explicitly out of scope per the brief, would be persistence: writing the roster to a JSON file on exit and reloading it on startup, so staff do not lose their list each time the program closes. The built-in `json` module would

suffice, but it would introduce a new validation concern (a missing, unreadable, or hand-edited save file) needing its own test cases, which is why it was scoped out rather than added under time pressure.

A "last updated" timestamp per device (via `datetime`) would let staff spot stale status entries: a device stuck on "under maintenance" for weeks is a signal worth surfacing automatically. Longer term, a delete/deactivate operation and a room-based filter would round out the feature set without changing the architecture, since both would be new `DeviceHub` methods following the same validated-method pattern, tested with the same equivalence-partitioning approach already used (Section 3.3).

A more architectural improvement would be separating the console-specific code into its own module distinct from the two classes, mirroring the layered structure where a front end can be swapped without touching the validated logic underneath. At this single-file scale that separation would add more structure than needed, but the design already anticipates it: `DeviceHub` has no `print()` or `input()` calls anywhere inside it.

Internationalisation was not considered: all prompts and status values are hard-coded English. Supporting other languages would be a reasonable next step for a non-English-speaking staff base; the closed `ALLOWED_STATUS` list keeps this a contained change, since every status string already flows through that one constant.

4.3 Self-Reflection

Building this project reinforced that the value of OOP at a small scale is not "more classes," it is discipline about where a rule lives. Every validation rule in this program exists in exactly one place, and every menu screen goes through it, which made writing the automated test suite considerably easier than it would have been if input checking were scattered across the console functions. It also clarified the practical difference between a logical error and a validation error: the former is a bug to be fixed, and the latter is expected user behaviour to be handled gracefully. Conflating the two early on led to over-defensive code before the design settled on raising `ValueError` deliberately for the second category.

Working through the flowchart-then-pseudocode-then-code sequence (Section 2.1) rather than opening an editor directly felt slowest at first, and turned out to save the most time: several edge cases in Section 3.3 (blank-versus-whitespace name, case-

insensitive duplicates) were spotted while sketching the flowchart, before a line of Python existed, so the corresponding checks were written correctly the first time.

The project also sharpened what "simple" should mean here. It would have been easy to read "simple system" as licence to skip validation, but the brief's own emphasis on edge cases and "both correct and incorrect user inputs" makes clear that simplicity applies to the *feature set*, not the *rigour* behind each feature. Holding those apart, few features, each properly validated, rather than many implemented loosely, is the biggest lesson this project reinforced.

Conclusion

The Smart Device Hub meets the four functional requirements set out by Bright Minds Academy (add, update status, view, and search) through a small, deliberately unglamorous OOP design: one class for a device, one class for the rules that govern a collection of them, and a thin console layer that never bypasses those rules. The automated self-check suite (19/19 passing) and the interactive transcript in Appendix A demonstrate that every validation path specified in the brief, both the valid and the invalid, behaves as designed. The result is a tool that is honest about its limits (no persistence, no real-time control) while being complete within the scope the brief actually asked for.

Revisiting the Introduction's framing of trust in data, the finished system's core claim is narrow but defensible: every record has passed through the same validated path, so staff reading the device list can rely on it meaning what it says. That narrowness is deliberate. Rather than a broad tool that half-handles many concerns, the project delivers a small tool that fully handles one, built on a design that keeps every rule behind that guarantee in exactly one place, and tested to demonstrate the guarantee holds.

References

- [1] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen, and K. A. Houston, **Object-Oriented Analysis and Design with Applications**, 3rd ed. Boston, MA: Addison-Wesley, 2007.
- [2] Python Software Foundation, "The Python Tutorial," **Python 3 Documentation**, 2024. [Online]. Available: <https://docs.python.org/3/tutorial/>
- [3] Python Software Foundation, "Classes," **Python 3 Documentation**, 2024. [Online]. Available: <https://docs.python.org/3/tutorial/classes.html>
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, **Design Patterns: Elements of Reusable Object-Oriented Software**. Boston, MA: Addison-Wesley, 1994.
- [5] I. Sommerville, **Software Engineering**, 10th ed. Harlow: Pearson Education, 2015.
- [6] R. S. Pressman and B. R. Maxim, **Software Engineering: A Practitioner's Approach**, 9th ed. New York: McGraw-Hill Education, 2019.
- [7] G. J. Myers, C. Sandler, and T. Badgett, **The Art of Software Testing**, 3rd ed. Hoboken, NJ: Wiley, 2011.
- [8] Python Software Foundation, "PEP 8 — Style Guide for Python Code," **Python Enhancement Proposals**, 2001. [Online]. Available: <https://peps.python.org/pep-0008/>
- [9] Python Software Foundation, "datetime — Basic date and time types," **Python 3 Documentation**, 2024. [Online]. Available: <https://docs.python.org/3/library/datetime.html>
- [10] B. W. Kernighan and R. Pike, **The Practice of Programming**. Reading, MA: Addison-Wesley, 1999.
- [11] ISO/IEC/IEEE 29119-1:2013, **Software and Systems Engineering — Software Testing — Part 1: Concepts and Definitions**, ISO/IEC/IEEE, 2013.

Appendices

Appendix A: Full interactive console transcript

Sequence exercised: add valid device → attempt blank-name add (rejected) → attempt duplicate-name add (rejected) → update status to a valid value → attempt invalid status value (rejected) → attempt status update on unknown device (rejected) → search for existing device → search for non-existent device (rejected) → view full device list → invalid menu choice → exit.

```
-----
SMART DEVICE HUB - Bright Minds Academy
-----

Devices registered: 0
-----

1. Add new device
2. Update device status
3. View all devices
4. Search for a device
5. Exit

Choose an option (1-5):
-- Add new device --
Device name: Smart Light A1
Room: Classroom 1
Added DEV-1: Smart Light A1 in Classroom 1 (status: offline)

Press Enter to continue...
-----

SMART DEVICE HUB - Bright Minds Academy
-----

Devices registered: 1
-----

Choose an option (1-5):
-- Add new device --
Device name:
Room:
Could not add device -> Device name is required.

Press Enter to continue...
-----

Choose an option (1-5):
```

```
-- Add new device --
Device name: Smart Light A1
Room: Classroom 1
Could not add device -> 'Smart Light A1' is already registered.

Press Enter to continue...
-----

Choose an option (1-5):
-- Update device status --

```

Tag	Name	Room	Status
DEV-1	Smart Light A1	Classroom 1	offline

```
-----

Device name or tag: DEV-1
Allowed statuses: online / offline / under maintenance
New status: online
Smart Light A1 is now 'online'.

Press Enter to continue...
-----

Choose an option (1-5):
-- Update device status --
Device name or tag: DEV-1
New status: notastatus
Could not update status -> Status must be one of: online / offline / under maintenance

Press Enter to continue...
-----

Choose an option (1-5):
-- Update device status --
Device name or tag: DEV-9
New status: online
Could not update status -> No device matches 'DEV-9'.

Press Enter to continue...
-----

Choose an option (1-5):
-- Search for a device --
Device name or tag: DEV-1

DEV-1 | Smart Light A1 | Room: Classroom 1 | Status: online
```

Press Enter to continue...

Choose an option (1-5):

-- Search for a device --

Device name or tag: nothere

Not found -> No device matches 'nothere'.

Press Enter to continue...

Choose an option (1-5):

-- All devices --

Tag	Name	Room	Status
-----	------	------	--------

DEV-1	Smart Light A1	Classroom 1	online
-------	----------------	-------------	--------

Press Enter to continue...

Choose an option (1-5): 9

Please enter a number from 1 to 5.

Press Enter to continue...

Choose an option (1-5): 5