



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

Assessment 01

Title:

Programming Assignment with Associated Documentation

Applied Computing.
Scripting and Programming Concepts
ACCA5036.

Dr. Koushik Modak (Lecturer)

University of Wales Trinity Saint David
(London Campus)

Student Name: Mr. Shugirthan Rasaratnam Amirthalingam

Student Number: 2430884

Group: 01

Team: 01

Date: 10/08/2026

Code Link :

https://github.com/Shugir/Scripting_and_Programming_Concepts_ACCA5036.git

Table of Contents

Introduction	3
Main Body	4
1. Analysis.....	4
1.1 Functional Requirements.....	4
1.2 Non-Functional Requirements	5
2. Software Design and Development	6
2.1 Design:	6
2.2 Implementation Decisions	10
2.3 Source Code.....	12
3. Testing and Debugging Report	30
3.1 The Importance of Programme Testing	30
3.2 Types of Errors and How They Were Fixed	31
3.3 Test Cases and Validation Evidence.....	32
4. Evaluation	40
4.1. Challenges Faced:.....	40
4.2 Future Improvements:	40
4.3 Self-Reflections:	41
Conclusion	41
References	42
Appendices	43
Appendix A: Automated Test Coverage Map.....	43
Appendix B: Quick Reference, Main Menu Options	44

Introduction

Bright Minds Academy is a small private tutoring centre piloting a smart classroom setup, where lights, projectors, thermostats and similar devices connect through a shared central system instead of being operated one by one. In principle that should make classrooms easier to run. In practice, staff on the pilot have found the opposite. Devices are currently tracked from memory and the odd handwritten note, so nobody can reliably say which projector is switched on, which room a particular smart light belongs to, or when a thermostat was last touched. As more devices get connected, that manual approach gets more fragile, and a faulty or forgotten device can go unnoticed until it disrupts a lesson.

This report covers the analysis, design, implementation, testing and evaluation of a Smart Home Device Manager: a lightweight, object-oriented Python console application built to give Bright Minds Academy staff one dependable place to register devices, change their status, and find them again quickly. The brief keeps the scope modest on purpose. There is no requirement for real-time device control, automation, or a persistent database, which matches the actual need at the Academy: a simple administrative record, not a home-automation platform. That modesty is a design constraint here, not a limitation to work around. The system stays intentionally small, and its quality is judged by how well it solves the stated problem rather than by how many features it accumulates.

Object-oriented programming was chosen as the implementation paradigm because the problem domain maps naturally onto it. A smart device really is a distinct 'thing' with its own identity, attributes (name, room, status, last-used time) and behaviour (changing status, matching a search query), and representing it as a class keeps that reasoning close to the code that implements it [1]. The rest of this report follows the development steps set out in the assignment brief. Section 1 analyses the business requirements and derives the functional and non-functional requirements. Section 2 covers the design (flowcharts and pseudocode) and the implementation decisions behind the data structures, control structures and modules used, followed by the annotated source code. Section 3 explains the testing strategy and shows evidence that the validation logic behaves correctly for valid and invalid input alike. Section 4 reflects on the challenges I ran into, proposes future improvements, and looks at what the project taught me as a developer.

Main Body

1. Analysis

Before any code was written, the business requirement (staff cannot easily tell what devices exist, where they are, or whether they are working) was broken down using abstraction and problem decomposition. Abstraction meant asking what a member of staff actually needs to know about a device, rather than everything that could theoretically be recorded about it, and that reduced a smart device to five attributes: a unique identifier, a name, a room, a status, and the time it was last changed. Problem decomposition was then applied at the architectural level. The single, fairly broad request 'manage smart devices' was split into five smaller, independent jobs: representing a device, enforcing the rules about what may be done to a device, reading and writing the console safely, presenting menus and screens, and starting the program. Each of those became its own Python module. That separation, covered in more detail in Section 2.2, meant every requirement below could be satisfied by a single, focused part of the system rather than by one large, tangled script.

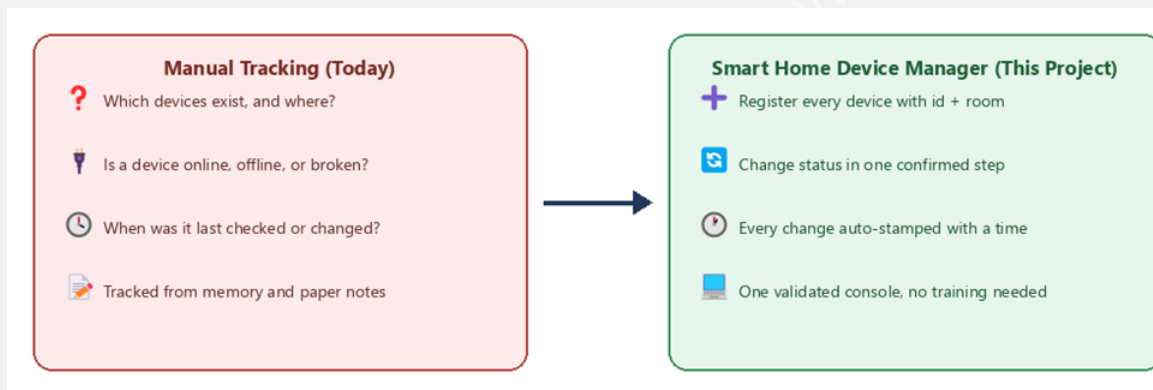


Figure A: The manual-tracking problem at Bright Minds Academy, and how each pain point maps onto a capability of the Smart Home Device Manager.

1.1 Functional Requirements

A features requirement analysis was carried out against the brief's four minimum operations, and three further operations were added where the pilot scenario clearly justified the extra effort. The functional requirements implemented in this version of the software are:

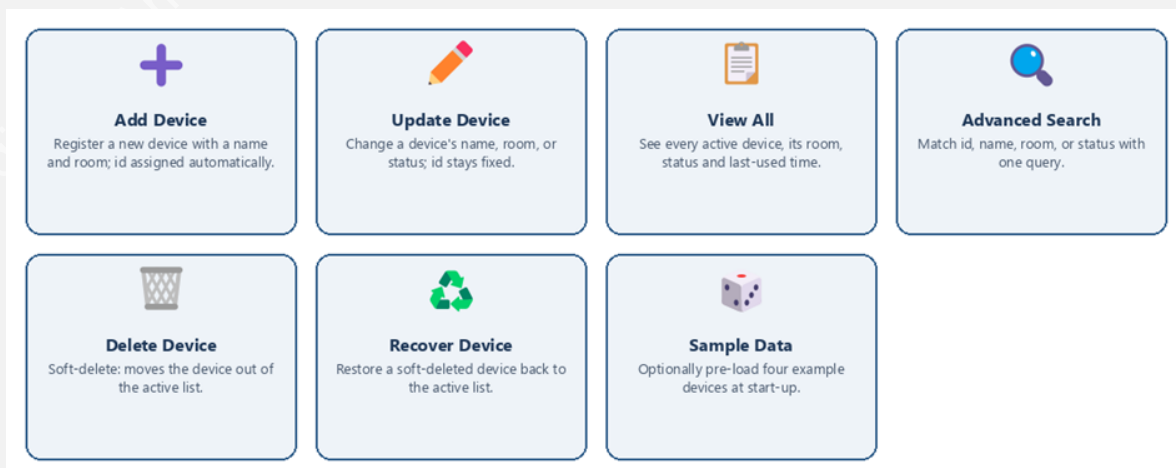


Figure B: The seven functional requirements implemented, four from the brief's minimum plus three added after requirements analysis.

- ❖ **Add a new device:** staff can register a device such as 'Smart Light A1' or 'Projector B2' together with the room it is located in. The system assigns a unique id automatically, so staff never have to invent or remember one.
- ❖ **Update a device:** goes beyond the brief's 'mark a device as online, offline, or under maintenance' to also allow renaming a device or moving it to a different room, since staff are just as likely to relocate a projector as to change its status.
- ❖ **View the device list:** staff can see every active device together with its room, status and the time it was last changed, in one screen.
- ❖ **Search for a device:** goes beyond a name-only lookup to match id, name, room or status in a single query, so a member of staff can type 'classroom 1' or 'offline' just as easily as a device name.
- ❖ **Delete a device:** added because a staff member decommissioning or mis-adding a device is a realistic scenario the brief's four operations do not cover on their own.
- ❖ **Recover a deleted device:** the safety net for the operation above. A delete only moves a device to a separate recoverable list; it never destroys the record.
- ❖ **Load sample data on start-up:** an optional first screen that pre-populates four example devices, so a marker (or a new staff member) can explore every menu option immediately without typing in test data by hand.

1.2 Non-Functional Requirements

In addition to functionality, the brief asks for a solution that is user-friendly, fast to load, and easy for non-technical staff to understand. These were translated into concrete quality criteria:



Figure C: The five non-functional quality criteria the system was designed against.

- ❖ **Usability:** a single numbered menu drives the whole program, every destructive action (delete) requires two separate confirmations, and the Esc key cancels out of any screen back to the main menu, so a hesitant or non-technical user never gets stuck mid-task.
- ❖ **Performance:** all data is held in memory as plain Python lists, so every operation (add, update, search) finishes in well under a millisecond for the handful of devices a single classroom pilot would

realistically register. The console itself launches in well under a second, meeting the brief's 'fast loading' requirement.

- ❖ **Reliability:** every user-facing operation is validated before it can change the data (Section 2.2), and deletion is soft and recoverable rather than destructive, so a mis-click cannot silently lose a device record.
- ❖ **Maintainability:** the system is split into five single-purpose modules (Section 2.2), so a future change to, say, how devices are displayed, cannot accidentally break the validation rules, and vice versa.
- ❖ **Portability:** the console runs unmodified on both Windows and Linux/macOS terminals by selecting the correct low-level keyboard-reading implementation at start-up (Section 2.2). That matters because Bright Minds Academy's classroom hardware is not guaranteed to be a single operating system.

2. Software Design and Development

This section documents how the requirements identified in Section 1 were turned into a working system: the algorithm design behind the core operations, the data structures and programming constructs chosen to implement them, and the annotated source code itself.

2.1 Design:

Two complementary tools were used to design the solution before implementation began: flowcharts, to show the overall control flow and decision points graphically, and pseudo-code, to pin down the exact logic of the core functions in a language-neutral form. Working through both first made the eventual Python implementation almost a direct translation exercise. It also surfaced a few edge cases early, such as what should happen if a user tries to rename a device to a name that is already taken, before any code needed rewriting to handle them.

2.1.1 Flowcharts:

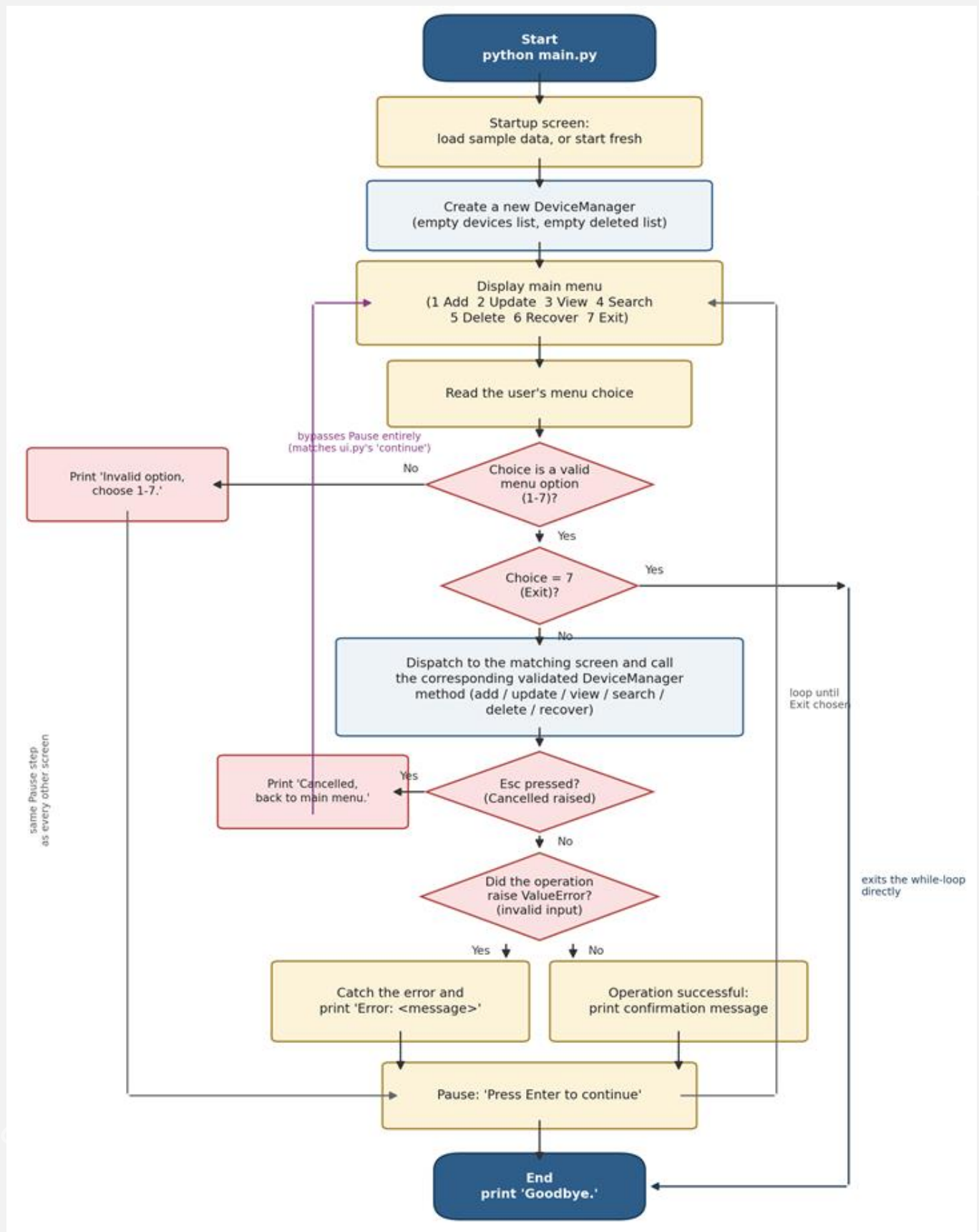


Figure 1: Overall application flow, the main menu loop that drives every screen.

Figure 1 shows the shape of the whole program: a single loop displays the menu, reads a choice, dispatches to the matching operation, and either reports an error or applies the change, before looping back

to the menu until the user chooses to exit. This structure maps directly onto the while loop and if/elif chain inside `run()` in `ui.py` (Section 2.3), and it is what keeps the 'Esc cancels back to the menu' behaviour consistent across every screen: every branch of the loop funnels back through the same pause-and-redisplay step.

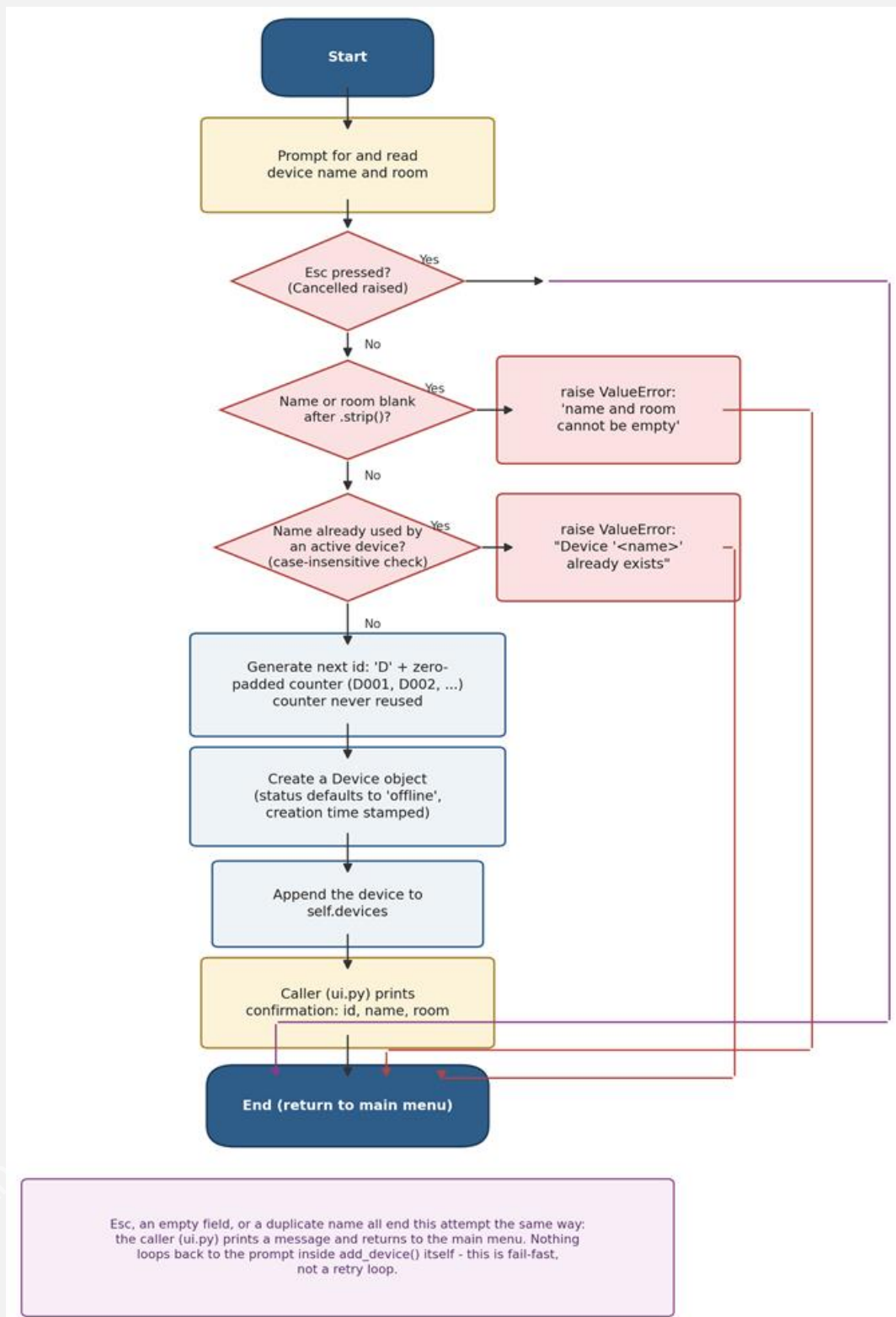


Figure 2: Detailed flow for the 'Add new device' operation, including both validation rules.

Figure 2 zooms into one core operation, adding a device, to show the validation logic that Figure 1 abstracts away as a single 'call the matching DeviceManager method' box. Two checks must both pass before a device is created: the name and room must not be blank once surrounding whitespace is stripped, and the name must not already belong to another active device. Either failure raises an error that sends the user back

to the input prompt instead of crashing the program, the same pattern used by every other write operation in the system (update, delete, recover).

2.1.2 Pseudo-code:

The three pseudo-code listings below cover the operations with the most decision logic: adding a device, updating a device, and searching. View, delete and recover follow the same shape as these three, so they are left out here for brevity; their full logic is visible in the annotated source code in Section 2.3.

```
FUNCTION add_device(name, room)
  name <- TRIM(name)
  room <- TRIM(room)
  IF name IS EMPTY OR room IS EMPTY THEN
    RAISE ValueError("Device name and room cannot be empty.")
  END IF
  IF an active device already has this name (case-insensitive) THEN
    RAISE ValueError("Device '" + name + "' already exists.")
  END IF
  new_id <- "D" + next counter value, zero-padded to 3 digits
  increment counter
  device <- NEW Device(new_id, name, room, status = "offline")
  APPEND device TO devices list
  RETURN device
END FUNCTION

FUNCTION update_device(device_id, new_name, new_room, new_status)
  device <- FIND device in devices list BY device_id
  IF device NOT FOUND THEN
    RAISE ValueError("Device id '" + device_id + "' not found.")
  END IF
  IF new_name IS NOT EMPTY THEN
    IF another active device already has new_name THEN
      RAISE ValueError("Device '" + new_name + "' already exists.")
    END IF
    device.name <- new_name
  END IF
  IF new_room IS NOT EMPTY THEN
    device.room <- new_room
  END IF
  IF new_status IS NOT EMPTY THEN
    IF new_status NOT IN {"online", "offline", "under maintenance"} THEN
      RAISE ValueError("Status must be one of the three allowed values.")
    END IF
    device.status <- new_status
  END IF
  device.last_used <- CURRENT_TIMESTAMP
  RETURN device
END FUNCTION

FUNCTION search(query)
  query <- TRIM(query)
  IF query IS EMPTY THEN
    RAISE ValueError("Search text cannot be empty.")
  END IF
  results <- EMPTY LIST
  FOR EACH device IN devices list
    IF query (lower-case) IS a substring of device.id, device.name,
      device.room, OR device.status THEN
      APPEND device TO results
    END IF
  END FOR
  IF results IS EMPTY THEN
    RAISE ValueError("No device matches '" + query + "'.")
  END IF
  RETURN results
END FUNCTION
```

2.2 Implementation Decisions

2.2.1 Data Structures:

The brief allows either lists or dictionaries. This solution uses a plain Python list of Device objects (self.devices in DeviceManager) rather than a list or dictionary of raw dictionaries, because the brief also asks for the system to be genuinely object-oriented and for classes to be used to 'handle objects'. A Device is defined as a class (Section 2.3) with five attributes (id, name, room, status and last_used) and two small behaviours of its own: set_status(), which updates status and the timestamp together, and matches(), which implements the substring search. A list, rather than a dictionary keyed by id, made sense here too: the number of devices a single classroom pilot will ever hold is small, so the O(n) linear scans used by _find_by_id() and _find_by_name() cost nothing noticeable in practice, and a list preserves a stable insertion order for the 'view all devices' screen without any extra sorting logic. A second list, self.deleted, stores soft-deleted devices using the same Device objects, which is how recover_device works without needing a separate 'undo' data structure. Finally, VALID_STATUSES is defined once as an immutable tuple in device.py and imported everywhere a status needs checking or displaying, so the three legal status strings ('online', 'offline', 'under maintenance') exist in exactly one place in the codebase.

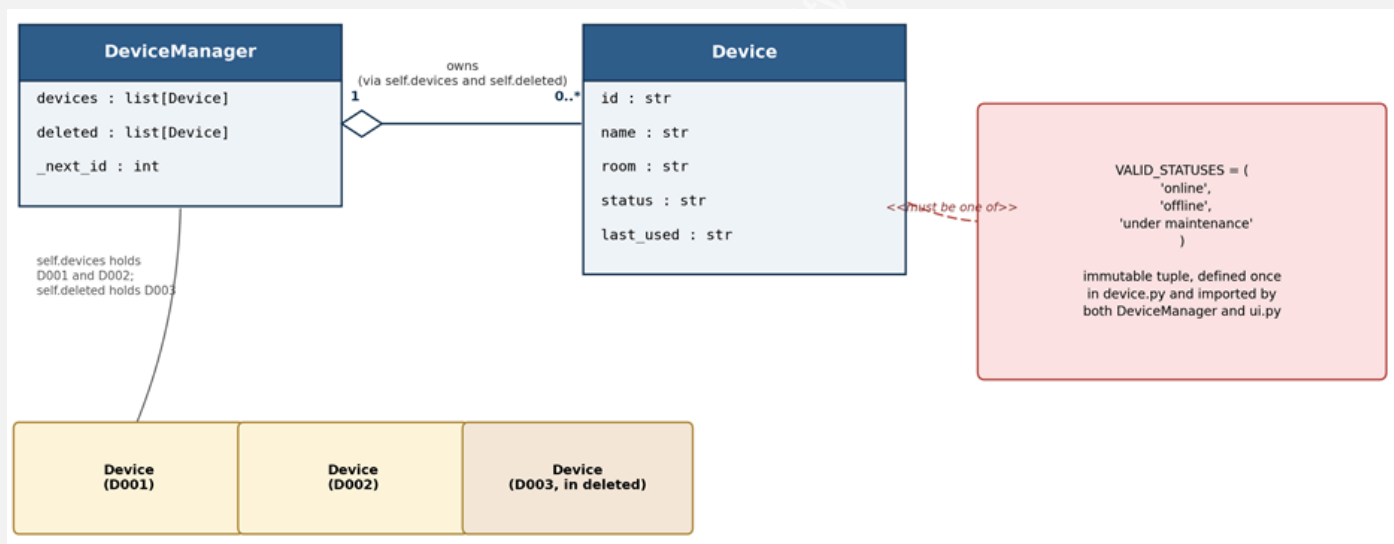


Figure D: How DeviceManager owns and organises its data: two lists of Device objects plus the shared VALID_STATUSES tuple.

2.2.2 Programming Constructs:

Control flow is kept simple and explicit throughout. The main screen dispatch in ui.py uses a single while True loop combined with an if/elif chain keyed on the user's menu choice. That was preferred over, say, a dictionary of callback functions, because the assignment brief specifically values code that is 'easy to understand', and an if/elif chain reads top-to-bottom exactly like the menu it implements. For loops are used wherever a full pass over the device list is genuinely required (search, the internal _find_by_* helpers). No while loop is used for iteration, since Python's for-in already expresses 'do this for every device' without a manual index or exit condition to get wrong. Every validation rule is implemented as an early return or raise at the top of a function rather than as nested if/else blocks, which keeps DeviceManager's methods close to a single level of indentation and easy to read as a checklist of preconditions.

Error handling is built entirely on Python's built-in exceptions rather than sentinel return values such as None or False. Every DeviceManager method that can fail raises ValueError with a specific, human-readable message, and ui.py catches ValueError in one place per screen and prints 'Error: <message>' [3]. That means a validation rule can never be silently ignored by a caller that forgets to check a return value; the error has to be either handled or allowed to propagate. A second, custom exception, Cancelled, is raised by read_line() when the user presses Esc, and it is used purely for control flow (abandon the current screen), keeping that concern separate from ValueError's job of reporting bad data. Functions are the only unit of reuse in this codebase; there is no inheritance hierarchy, because a single Device class with no meaningful subtypes did not need one. The OOP requirement is met instead through encapsulation (Device and DeviceManager each own and protect their own state) and abstraction (callers of DeviceManager never touch self.devices directly) [1], [10].

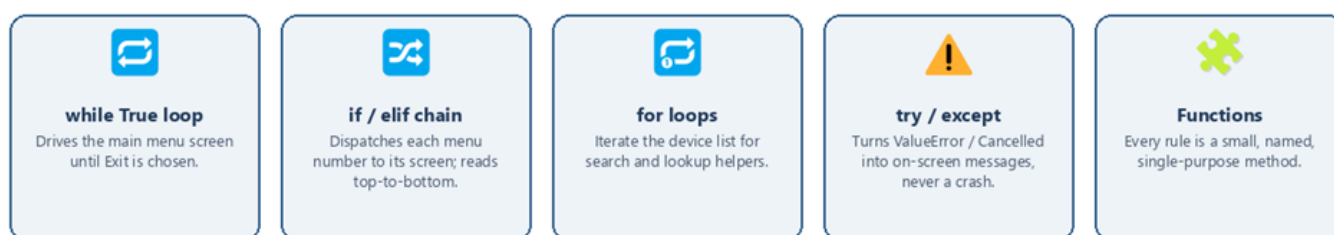


Figure E: The five programming constructs the codebase relies on, and where each one is used.

2.2.3 Python Modules and Libraries:

The project uses only the Python standard library, with no third-party dependencies to install. That directly supports the brief's 'fast loading' and 'easy to understand' requirements: anyone with a plain Python interpreter can run it immediately.

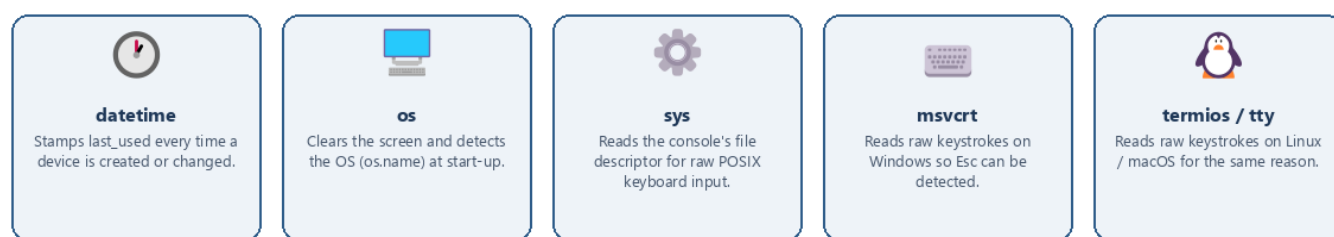


Figure F: Every standard-library module used in the project, and what it is responsible for.

- ❖ **datetime**: used in device.py and device_manager.py to stamp last_used with a human-readable timestamp every time a device is created or changed, giving staff a simple audit trail of when each record last moved [2].
- ❖ **os**: used in console_io.py both to clear the screen (os.system('cls' or 'clear')), so each screen redraws in place instead of scrolling, and to detect the operating system (os.name) so the correct keyboard-reading implementation gets picked.
- ❖ **sys**: used to obtain the console's file descriptor (sys.stdin.fileno()) on POSIX systems, needed to put the terminal into character-at-a-time mode.

- ❖ **msvcrt**: the Windows-only module used to read raw keystrokes one at a time instead of through Python's line-buffered input(). This is what makes Esc-to-cancel possible on Windows.
- ❖ **termios / tty**: the POSIX equivalent of msvcrt, used on Linux/macOS for the same raw keystroke reason.

The project uses no file, database, or networking module anywhere, by design, matching the brief's statement that persistence is out of scope for this version.

2.3 Source Code

The full, working source code is reproduced below exactly as it runs, organised across five files that mirror the architecture described in Section 2.2. Every file begins with a module-level docstring explaining its responsibility and how it fits with the others, and every non-obvious decision inside a function has a short comment explaining why that choice was made, following the general guidance that comments should explain intent rather than restate the code [4], [7]. A live, fully functional copy of this code is hosted in the following GitHub repository:

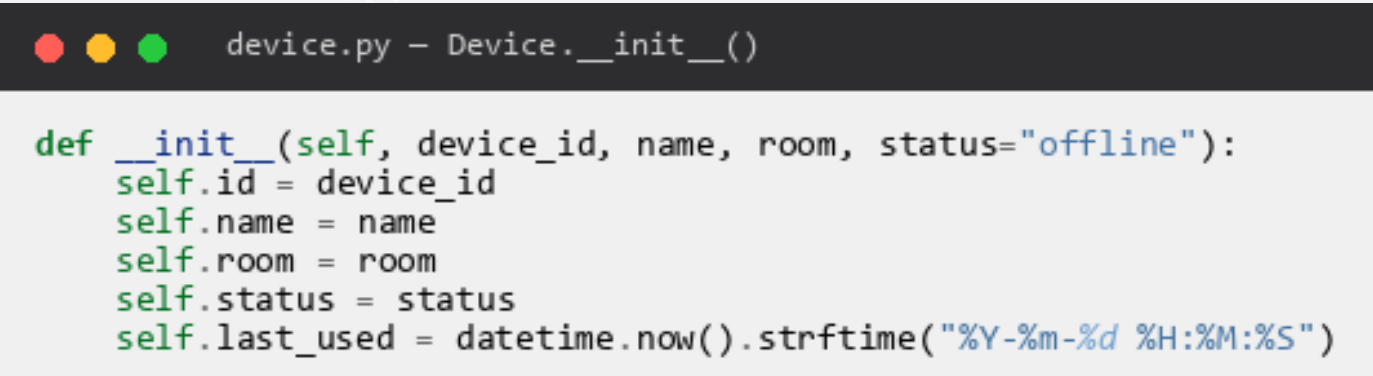
https://github.com/Shugir/Scripting_and_Programming_Concepts_ACCA5036.git

Each file below is presented method by method: a clean screenshot of that method's code (with internal comments and docstrings removed, so the logic itself is easy to scan) followed by a short explanation of what it does and why it was written that way. The commented, fully annotated version of every file is the one actually running in the GitHub repository linked above.

2.3.1 device.py

Data model: one smart device record. No console I/O, kept 'dumb' so it can be reused or unit-tested without dragging in any CLI code.

Device.__init__()



```
def __init__(self, device_id, name, room, status="offline"):
    self.id = device_id
    self.name = name
    self.room = room
    self.status = status
    self.last_used = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
```

Sets up a new Device with five attributes: an id assigned once by DeviceManager and never changed afterwards, the name and room supplied by the caller, a status that defaults to 'offline' if none is given, and a last_used timestamp stamped immediately with datetime.now(). Even a freshly created device already has an audit-trail entry.

Device.set_status()

```
device.py - Device.set_status()

def set_status(self, status):
    self.status = status
    self.last_used = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
```

Updates the status field and, in the same call, refreshes last_used to the current time. Bundling both changes into one method means the timestamp can never fall out of sync with the status it describes, because nothing changes one without the other.

Device.matches()

```
device.py - Device.matches()

def matches(self, query):
    query = query.strip().lower()
    fields = (self.id, self.name, self.room, self.status)
    return any(query in str(f).lower() for f in fields)
```

Implements the substring search behind 'advanced search'. The query is trimmed and lower-cased once, then checked against id, name, room and status in a single any(...) expression, so a query such as 'classroom 1' can match on any field.

Device.__str__()

```
device.py - Device.__str__()

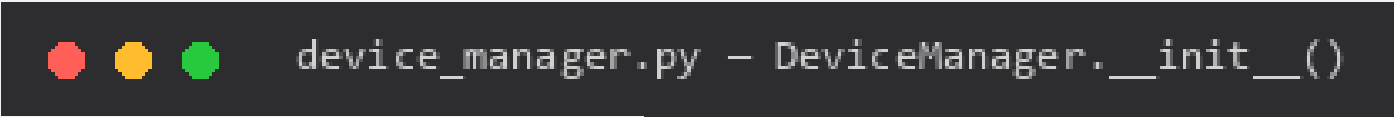
def __str__(self):
    return (
        f"{self.id:<6} | {self.name:<20} | {self.room:<15} | "
        f"{self.status:<18} | {self.last_used}"
    )
```

Formats a device as one fixed-width row (id, name, room, status, last_used). That lets print_table() in ui.py print an aligned table just by looping and calling print(device), with no per-screen formatting function needed.

2.3.2 device_manager.py

Business logic layer: owns the device list and every validated operation on it (add, update, delete, recover, search). This is where every validation rule discussed in Section 2.1/2.2 actually lives.

DeviceManager.__init__()



```
device_manager.py - DeviceManager.__init__()

def __init__(self):
    self.devices = []
    self.deleted = []
    self._next_id = 1
```

Creates an empty manager: an empty devices list, an empty deleted list, and a `_next_id` counter starting at 1 that never decreases, so every id ever issued stays unique for the life of the running program.

DeviceManager._find_by_id()



```
device_manager.py - DeviceManager._find_by_id()

def _find_by_id(self, device_id):
    device_id = device_id.strip().lower()
    for d in self.devices:
        if d.id.lower() == device_id:
            return d
    return None
```

Internal helper that scans `self.devices` for a matching id, case-insensitively, so 'd001' and 'D001' both resolve to the same record. It returns `None` rather than raising, which leaves the calling public method to decide how to report a miss.

DeviceManager._find_by_name()

 device_manager.py – DeviceManager._find_by_name()

```
def _find_by_name(self, name, exclude=None):
    name = name.strip().lower()
    for d in self.devices:
        if d is not exclude and d.name.lower() == name:
            return d
    return None
```

Internal helper that enforces 'no two active devices share a name'. Its optional exclude parameter lets update_device() check every other device for a clash while still letting a device keep its own current name during an edit.

DeviceManager.add_device()

 device_manager.py – DeviceManager.add_device()

```
def add_device(self, name, room):
    name = name.strip()
    room = room.strip()
    if not name or not room:
        raise ValueError("Device name and room cannot be empty.")
    if self._find_by_name(name):
        raise ValueError(f"Device '{name}' already exists.")
    device_id = f"D{self._next_id:03d}"
    self._next_id += 1
    device = Device(device_id, name, room)
    self.devices.append(device)
    return device
```

Validates that name and room are non-empty after trimming, checks the name is not already taken, generates the next sequential id (D001, D002, and so on), creates the Device, appends it, and returns it. Figure 2's flowchart maps directly onto this method.

DeviceManager.update_status()

```
device_manager.py - DeviceManager.update_status()

def update_status(self, device_id, status):
    status = status.strip().lower()
    if status not in VALID_STATUSES:
        raise ValueError(f"Status must be one of {VALID_STATUSES}.")
    device = self._find_by_id(device_id)
    if not device:
        raise ValueError(f"Device id '{device_id}' not found.")
    device.set_status(status)
    return device
```

A narrow, single-purpose update used by `load_sample_data()`. It validates the new status against `VALID_STATUSES`, finds the device, and calls its `set_status()`. It stays separate from `update_device()` so that call site reads as a clear, one-line intent.

DeviceManager.update_device()

```
device_manager.py - DeviceManager.update_device()

def update_device(self, device_id, new_name=None, new_room=None, new_status=None):
    device = self._find_by_id(device_id)
    if not device:
        raise ValueError(f"Device id '{device_id}' not found.")

    if new_name:
        new_name = new_name.strip()
        if self._find_by_name(new_name, exclude=device):
            raise ValueError(f"Device '{new_name}' already exists.")
        device.name = new_name

    if new_room:
        device.room = new_room.strip()

    if new_status:
        new_status = new_status.strip().lower()
        if new_status not in VALID_STATUSES:
            raise ValueError(f"Status must be one of {VALID_STATUSES}.")
        device.status = new_status

    device.last_used = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    return device
```

The general-purpose edit behind the 'Update device' screen. Each of `new_name`, `new_room` and `new_status` is optional, so a blank value leaves that field untouched. A rename still gets checked for a name clash, the status still gets validated, and the id itself is never editable here, since it is the one stable handle a user has to a device.

DeviceManager.delete_device()

```
device_manager.py - DeviceManager.delete_device()

def delete_device(self, device_id):
    device = self._find_by_id(device_id)
    if not device:
        raise ValueError(f"Device id '{device_id}' not found.")
    self.devices.remove(device)
    self.deleted.append(device)
    return device
```

Performs a soft delete: finds the device, removes it from `self.devices`, and appends it to `self.deleted` instead of discarding it. Nothing is actually thrown away, which is exactly what `recover_device()` below relies on.

DeviceManager._find_deleted_by_id()

```
device_manager.py - DeviceManager._find_deleted_by_id()

def _find_deleted_by_id(self, device_id):
    device_id = device_id.strip().lower()
    for d in self.deleted:
        if d.id.lower() == device_id:
            return d
    return None
```

The same lookup pattern as `_find_by_id()`, just scoped to `self.deleted`, and used only by `recover_device()`.

DeviceManager.recover_device()

```
device_manager.py - DeviceManager.recover_device()

def recover_device(self, device_id):
    device = self._find_deleted_by_id(device_id)
    if not device:
        raise ValueError(f"Deleted device id '{device_id}' not found.")
    if self._find_by_name(device.name):
        raise ValueError(
            f"Cannot recover: an active device named '{device.name}' already exists."
        )
    self.deleted.remove(device)
    self.devices.append(device)
    return device
```

Restores a soft-deleted device back into `self.devices`, but first checks whether an active device has since taken the same name. If one has, the recovery gets rejected instead of silently creating a duplicate, an edge case that was actually caught while sketching the design in Section 2.1, not while coding.

DeviceManager.list_devices()

```
device_manager.py – DeviceManager.list_devices()
```

```
def list_devices(self):  
    return list(self.devices)
```

Returns a shallow copy of the active device list, so a caller in ui.py can loop over it or check its length without being able to mutate DeviceManager's internal state by accident.

DeviceManager.list_deleted()

```
device_manager.py – DeviceManager.list_deleted()
```

```
def list_deleted(self):  
    return list(self.deleted)
```

The same copy-returning pattern as list_devices(), just scoped to the soft-deleted list, used by the 'Recover deleted device' screen.

DeviceManager.get_by_id()

```
device_manager.py – DeviceManager.get_by_id()
```

```
def get_by_id(self, device_id):  
    device = self._find_by_id(device_id)  
    if not device:  
        raise ValueError(f"Device id '{device_id}' not found.")  
    return device
```

A public wrapper around _find_by_id() that raises ValueError with a clear message instead of returning None. Every screen in ui.py can then rely on catching a single exception type rather than remembering to check for None.

DeviceManager.search()

   device_manager.py – DeviceManager.search()

```
def search(self, query):
    query = query.strip()
    if not query:
        raise ValueError("Search text cannot be empty.")
    results = [d for d in self.devices if d.matches(query)]
    if not results:
        raise ValueError(f"No device matches '{query}'.")
    return results
```

Implements 'advanced search': checks the query is non-empty, filters self.devices using each device's own matches(), and raises ValueError if nothing matched instead of silently returning an empty list. That way a typo gets reported clearly rather than just looking like the device list is empty.

2.3.3 console_io.py

Low-level, platform-specific console helpers: screen clearing and the Esc-cancellable raw-keystroke reader that input() cannot provide.

clear_screen()

   console_io.py – clear_screen()

```
def clear_screen():
    os.system("cls" if os.name == "nt" else "clear")
```

Clears the terminal with the correct platform command (cls on Windows, clear elsewhere), so every screen redraws in the same spot instead of the output scrolling endlessly.



```
def read_line(prompt=""):
    print(prompt, end="", flush=True)
    chars = []
    while True:
        ch = msvcrt.getch()
        if ch == b"\x1b":
            print()
            raise Cancelled()
        if ch in (b"\r", b"\n"):
            print()
            return "".join(chars)
        if ch == b"\x08":
            if chars:
                chars.pop()
                print("\b \b", end="", flush=True)
            continue
        try:
            c = ch.decode("utf-8")
        except UnicodeDecodeError:
            continue
        chars.append(c)
        print(c, end="", flush=True)
```

The Windows implementation of Esc-cancellable input. `msvcrt.getch()` reads one raw keystroke directly from the console, which lets this function detect Esc (raises `Cancelled`), Enter (returns the typed text) and Backspace (edits the in-progress line), none of which Python's built-in `input()` can do.

read_line(), Linux / macOS (termios / tty)

```
console_io.py - read_line()

def read_line(prompt=""):
    print(prompt, end="", flush=True)
    chars = []
    fd = sys.stdin.fileno()
    old_settings = termios.tcgetattr(fd)
    try:
        tty.setcbreak(fd)
        while True:
            ch = sys.stdin.read(1)
            if ch == ESC:
                print()
                raise Cancelled()
            if ch in ("\r", "\n"):
                print()
                return "".join(chars)
            if ch == "\x7f":
                if chars:
                    chars.pop()
                    print("\b \b", end="", flush=True)
                continue
            chars.append(ch)
            print(ch, end="", flush=True)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
```

The Linux/macOS equivalent of the function above. `tty.setcbreak()` temporarily switches the terminal out of line-buffered mode so each keystroke can be read immediately, and the finally block makes sure normal terminal settings come back even if the function exits via `Cancelled`.

pause()

 console_io.py – pause()

```
def pause():  
    try:  
        read_line("\nPress Enter to continue...")  
    except Cancelled:  
        pass
```

A shared 'press Enter to continue' step used after every screen. It calls `read_line()` but ignores a `Cancelled` exception, since there's nothing to cancel here: Enter and Esc both lead back to the main menu anyway.

2.3.4 ui.py

Menu screens and the main loop. The only file that prints menus or reads keyboard input for the running application, and it depends on `device_manager.py` for every rule.

print_header()

 ui.py – print_header()

```
def print_header(manager):  
    print("=" * 55)  
    print("      SMART HOME DEVICE MANAGER - Bright Minds Academy")  
    print("=" * 55)  
    print(f"Devices tracked: {len(manager.devices)}\n")
```

Draws the banner shown at the top of every screen, including a live count of tracked devices, so the user always has an at-a-glance sense of how much data currently exists.

print_menu()

```
ui.py - print_menu()

def print_menu():
    print("1. Add new device")
    print("2. Update device (name, room, or status)")
    print("3. View all devices")
    print("4. Advanced search (by id, name, room, or status)")
    print("5. Delete device")
    print("6. Recover deleted device")
    print("7. Exit")
```

Prints the seven numbered main menu options. It stays its own tiny function so the numbers here and the if/elif chain in run() can be kept in sync by inspection.

print_table()

```
ui.py - print_table()

def print_table(devices):
    print(f"{'ID':<6} | {'Name':<20} | {'Room':<15} | {'Status':<18} | Last Used")
    print("-" * 83)
    for d in devices:
        print(d)
```

The single, shared table renderer used by View all, Search results, Delete and Recover, so results always line up the same way no matter which screen called it.

load_sample_data()

 ui.py – load_sample_data()

```
def load_sample_data(manager):
    sample = [
        ("Smart Light A1", "Living Room", "online"),
        ("Projector B2", "Classroom 1", "offline"),
        ("Thermostat C3", "Office", "under maintenance"),
        ("Smart Light D4", "Classroom 2", "online"),
    ]
    for name, room, status in sample:
        device = manager.add_device(name, room)
        manager.update_status(device.id, status)
```

Pre-populates a fresh DeviceManager with four example devices covering all three statuses, so a marker, or a new staff member, can explore every menu option straight away.

startup_screen()

 ui.py – startup_screen()

```
def startup_screen():
    manager = DeviceManager()
    while True:
        clear_screen()
        print_header(manager)
        print("-- Welcome --\n")
        print("1. Load with sample data")
        print("2. Real application (start fresh)")
        choice = input("\nChoose an option (1-2): ").strip()
        if choice == "1":
            load_sample_data(manager)
            return manager
        elif choice == "2":
            return manager
        else:
            print("\nInvalid option, choose 1-2.")
            pause()
```

The first screen shown at launch: choose sample data or start empty. The DeviceManager itself gets created here, once per run, matching the decision not to persist data between sessions.

update_device_wizard()

```
ui.py - update_device_wizard()

def update_device_wizard(manager, device):
    print(f"\nCurrent: {device}")
    print(f"Id: {device.id} (cannot be changed)")
    print("Leave a field blank to keep its current value. Press Esc to cancel.")
    new_name = read_line(f"New name [{device.name}]: ")
    new_room = read_line(f"New room [{device.room}]: ")
    print(f"Valid statuses: {' '.join(VALID_STATUSES)}")
    new_status = read_line(f"New status [{device.status}]: ")
    updated = manager.update_device(device.id, new_name, new_room, new_status)
    print(f"\nUpdated: {updated}")
```

Prompts for a new name, room and status one field at a time (a blank keeps the current value), reminds the user the id cannot change, and calls `manager.update_device()`. Both the direct 'Update device' option and the 'edit a search result' shortcut share this one function.

delete_device_wizard()

```
ui.py - delete_device_wizard()

def delete_device_wizard(manager, device):
    print(f"\nAbout to delete: {device}")
    confirm1 = read_line("Are you sure you want to delete this device? (y/n): ").strip().lower()
    if confirm1 != "y":
        print("\nDelete cancelled.")
        return
    confirm2 = read_line(f"Type the device id '{device.id}' to confirm: ").strip()
    if confirm2.lower() != device.id.lower():
        print("\nId did not match. Delete cancelled.")
        return
    manager.delete_device(device.id)
    print(f"\nDeleted: {device.id} - {device.name}. (recoverable via 'Recover deleted device')")
```

Implements the double-confirmation delete flow: a y/n prompt, then having to correctly retype the exact device id. That way an accidental keypress alone cannot delete anything.

run() : startup + main loop

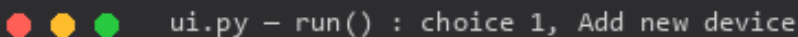


```
def run():
    manager = startup_screen()

    while True:
        clear_screen()
        print_header(manager)
        print_menu()
        choice = input("\nChoose an option (1-7): ").strip()
```

The outer shape of the whole application: get a DeviceManager from startup_screen(), then loop forever, redrawing the header and menu and reading one choice each time round, the loop visualised in Figure 1.

run() : choice 1, Add new device



```
if choice == "1":
    clear_screen()
    print_header(manager)
    print("-- Add new device -- (Esc to cancel)\n")
    try:
        name = read_line("Device name: ")
        room = read_line("Room: ")
        device = manager.add_device(name, room)
        print(f"\nDevice added: {device.id} - {device.name} ({device.room}).")
    except Cancelled:
        print("\nCancelled, back to main menu.")
        continue
    except ValueError as e:
        print(f"\nError: {e}")
    pause()
```

Choice 1 reads a name and room, calls manager.add_device(), and reports either the new id or the validation error, the flow detailed in Figure 2.

run() : choice 2, Update device

```
ui.py - run() : choice 2, Update device

elif choice == "2":
    clear_screen()
    print_header(manager)
    print("-- Update device --\n")
    if manager.devices:
        print_table(manager.list_devices())
    try:
        device_id = read_line("\nDevice id to update (Esc to cancel): ")
        device = manager.get_by_id(device_id)
        update_device_wizard(manager, device)
    except Cancelled:
        print("\nCancelled, back to main menu.")
        continue
    except ValueError as e:
        print(f"\nError: {e}")
    pause()
```

Choice 2 shows the current device table for reference, reads the id to edit, looks it up, and hands off to `update_device_wizard()` to collect and apply the new values.

run() : choice 3, View all devices

```
ui.py - run() : choice 3, View all devices

elif choice == "3":
    clear_screen()
    print_header(manager)
    print("-- All devices --\n")
    devices = manager.list_devices()
    if not devices:
        print("No devices yet.")
    else:
        print_table(devices)
    pause()
```

Choice 3 is the simplest screen in the program: list the active devices via `print_table()`, or say plainly that there are none yet.

run() : choice 4, Advanced search

```
● ● ● ui.py - run() : choice 4, Advanced search

elif choice == "4":
    clear_screen()
    print_header(manager)
    print("-- Advanced search -- (Esc to cancel)\n")
    try:
        query = read_line("Search by id, name, room, or status: ")
        results = manager.search(query)
        print(f"\n{len(results)} match(es) found:\n")
        print_table(results)
        pick = read_line(
            "\nEnter a device id to update it, or press Enter for main menu: "
        )
        if pick.strip():
            device = manager.get_by_id(pick)
            update_device_wizard(manager, device)
    except Cancelled:
        print("\nCancelled, back to main menu.")
        continue
    except ValueError as e:
        print(f"\nError: {e}")
    pause()
```

Choice 4 reads a search query, shows every match via `print_table()`, and offers a shortcut straight into `update_device_wizard()` if the user types one of the matching ids.

run() : choice 5, Delete device

```
● ● ● ui.py - run() : choice 5, Delete device

elif choice == "5":
    clear_screen()
    print_header(manager)
    print("-- Delete device -- (Esc to cancel)\n")
    if manager.devices:
        print_table(manager.list_devices())
    else:
        print("No devices yet.")
    try:
        device_id = read_line("\nDevice id to delete: ")
        device = manager.get_by_id(device_id)
        delete_device_wizard(manager, device)
    except Cancelled:
        print("\nCancelled, back to main menu.")
        continue
    except ValueError as e:
        print(f"\nError: {e}")
    pause()
```

Choice 5 shows the active device table, reads the id to remove, and hands off to `delete_device_wizard()` for the double-confirmation flow.

run() : choice 6, Recover deleted device

```
● ● ● ui.py - run() : choice 6, Recover deleted device

elif choice == "6":
    clear_screen()
    print_header(manager)
    print("-- Recover deleted device -- (Esc to cancel)\n")
    deleted = manager.list_deleted()
    if not deleted:
        print("No deleted devices to recover.")
    else:
        print_table(deleted)
        try:
            device_id = read_line("\nDevice id to recover: ")
            device = manager.recover_device(device_id)
            print(f"\nRecovered: {device.id} - {device.name}.")
        except Cancelled:
            print("\nCancelled, back to main menu.")
            continue
        except ValueError as e:
            print(f"\nError: {e}")
    pause()
```

Choice 6 shows the soft-deleted device table (or says it is empty), reads an id, and calls `manager.recover_device()`, reporting success or error like every other screen.

run() : choice 7 (Exit) and invalid choice

```
● ● ● ui.py - run() : choice 7 (Exit) and invalid choice

elif choice == "7":
    clear_screen()
    print("Goodbye.")
    break

else:
    print("\nInvalid option, choose 1-7.")
    pause()
```

Choice 7 ends the program cleanly with a goodbye message. Any other input falls through to the else branch, which just asks the user to choose again instead of crashing.

2.3.5 main.py

Entry point. Kept tiny on purpose, so there is one obvious place to run the program from.

main.py (whole file, no functions)

```
main.py

"""Entry point for the Smart Home Device Manager. Run this file to start the app.

Kept deliberately tiny: all real logic lives in device.py, device_manager.py,
console_io.py, and ui.py. This file exists only so there's one obvious
place to run (`python main.py`) and one obvious place to look first when
exploring the project.
"""

from ui import run

if __name__ == "__main__":
    run()
```

Importing and calling ui.run() is the only thing this file does. Keeping it this small leaves exactly one obvious place to start reading the codebase, and exactly one obvious place to run it from: python main.py.

3. Testing and Debugging Report

3.1 The Importance of Programme Testing

Testing is what turns 'the code ran once on my machine' into a real claim that the software behaves correctly. For a system whose entire purpose is to be a trustworthy record for staff who are not developers, that distinction matters more than usual. An unvalidated add_device() or update_status() would let the register quietly fill up with bad data that nobody notices until it causes confusion in a real classroom. Testing incorrect input on purpose, not just correct input, is what actually exercises the validation logic in Section 2.2. A suite that only ever supplies valid data would never prove a single ValueError check works [8], [9].

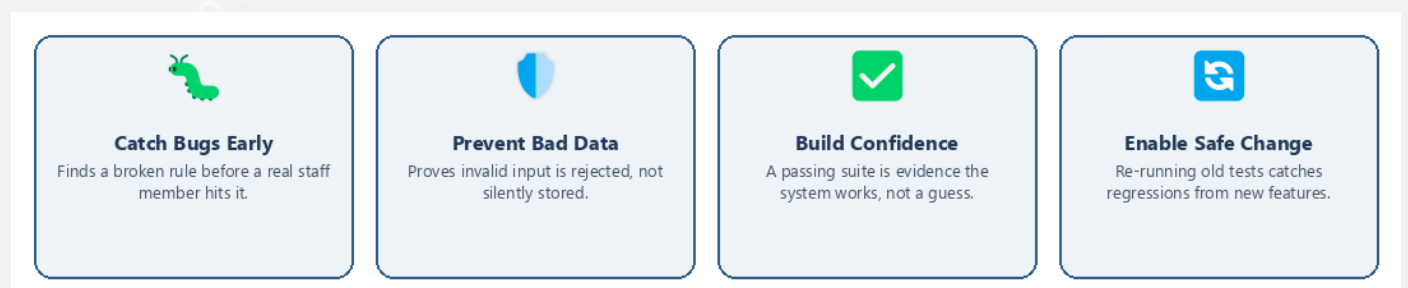
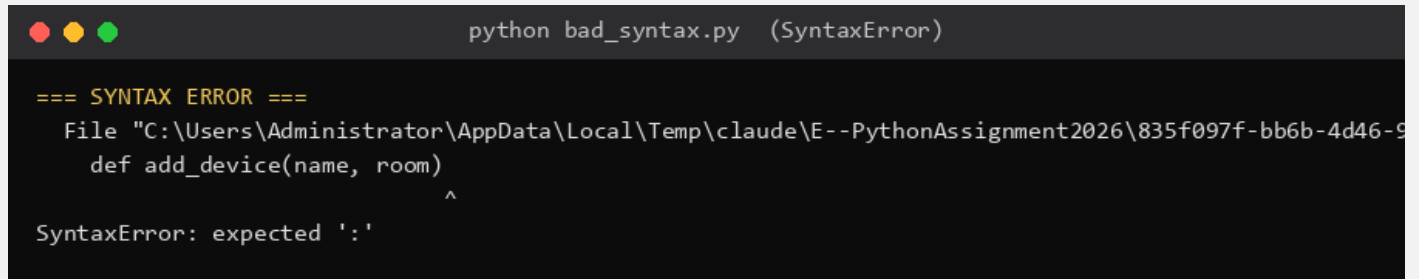


Figure G: Why this project treats testing as a required deliverable, not an afterthought.

3.2 Types of Errors and How They Were Fixed

Three broad categories of error appeared during development:

- ❖ Syntax errors: code Python cannot even parse, such as a missing colon or a mismatched bracket. These are the easiest to resolve, since Python refuses to run the file at all and reports the exact line and column, so the fix is just to correct that line and re-run. Figure H reproduces a real example.

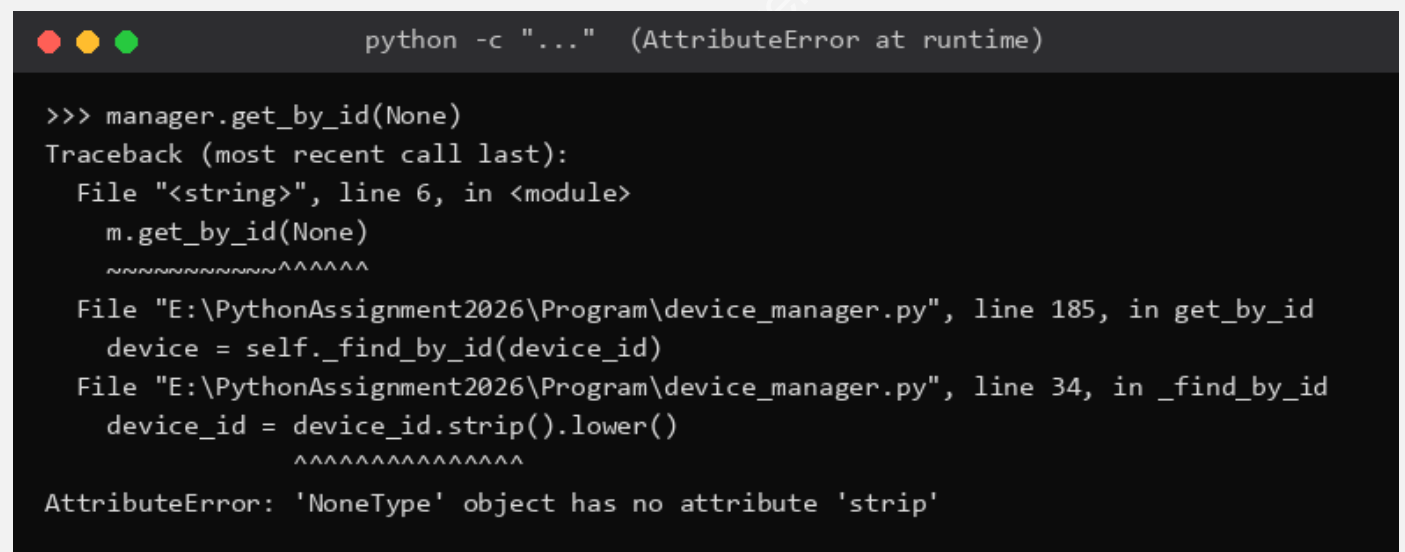


```
python bad_syntax.py (SyntaxError)

=== SYNTAX ERROR ===
File "C:\Users\Administrator\AppData\Local\Temp\claude\E--PythonAssignment2026\835f097f-bb6b-4d46-9...
    def add_device(name, room)
                        ^
SyntaxError: expected ':'
```

Figure H: A genuine `SyntaxError`, captured by intentionally omitting a colon from a function definition.

- ❖ Runtime errors: raised while the program is executing, once it has already been accepted by the parser. Figure I shows a real one produced by this codebase. Calling `get_by_id(None)` reaches `device_id.strip()` inside `_find_by_id()` and fails, because `None` has no `.strip()` method. This class of error gets caught by exercising every code path with a range of inputs, including ones a well-behaved caller would never actually pass.



```
python -c "..." (AttributeError at runtime)

>>> manager.get_by_id(None)
Traceback (most recent call last):
  File "<string>", line 6, in <module>
    m.get_by_id(None)
    ~~~~~^~~~~~
  File "E:\PythonAssignment2026\Program\device_manager.py", line 185, in get_by_id
    device = self._find_by_id(device_id)
  File "E:\PythonAssignment2026\Program\device_manager.py", line 34, in _find_by_id
    device_id = device_id.strip().lower()
                ^^^^^^^^^^^^^^^^^
AttributeError: 'NoneType' object has no attribute 'strip'
```

Figure I: A genuine `AttributeError` raised by `DeviceManager.get_by_id()` when called with `None` instead of a string.

- ❖ Logical errors: the program keeps running but produces the wrong result, which makes this the most dangerous category, since nothing crashes to draw attention to it. One turned up during manual testing: comparing device names for duplicates with a plain `==` would have let 'Smart Light A1' and 'smart light a1' coexist as two 'different' devices. The fix, in `_find_by_name()`, was to compare `name.strip().lower()` on both sides consistently, the same pattern used for ids and statuses throughout `DeviceManager`. Figure J is real, current output proving the fixed comparison now treats both spellings as the same device.

```
verifying the case-insensitive duplicate-name fix

>>> manager.add_device('smart light a1', 'Bedroom') # different case, same name
Error: Device 'smart light a1' already exists.

name.strip().lower() comparison correctly treats
'Smart Light A1' and 'smart light a1' as the SAME device name.
```

Figure J: Verifying the case-insensitive duplicate-name fix. 'smart light a1' is correctly rejected as a duplicate of the existing 'Smart Light A1'.

In general, syntax errors got caught immediately by the interpreter, runtime errors got caught by exercising every keyboard-driven code path by hand with a range of inputs, and logical errors got caught by writing down, in advance, what each function should do for a given input (Section 2.1) and then checking the actual output against that expectation. That is the same discipline the automated test suite in Section 3.3 formalises.

3.3 Test Cases and Validation Evidence

Two layers of evidence are presented below for every DeviceManager method that enforces a validation rule: the automated assertions that exercise it directly, and a screenshot of the same behaviour through the real menu screens a member of staff would actually use. Both layers drive DeviceManager directly rather than through console_io.py's read_line(), because that function needs a real interactive terminal and cannot be fed simulated keyboard input the way a normal function call can [12]. Testing DeviceManager on its own is not a compromise here, since none of the validation logic in Section 2.2 lives in ui.py anyway.

add_device()

```
test_smart_device_manager.py - demo() : add_device checks

m = DeviceManager()

d1 = m.add_device("Smart Light A1", "Room 1")
assert d1.id == "D001"
d2 = m.add_device("Projector B2", "Room 2")
assert d2.id == "D002"
assert len(m.devices) == 2

try:
    m.add_device("Smart Light A1", "Room 3")
    assert False, "duplicate name should have raised"
except ValueError:
    pass

try:
    m.add_device(" ", "Room 1")
    assert False, "empty name should have raised"
except ValueError:
    pass
```

```
python main.py -- Add new device

=====
SMART HOME DEVICE MANAGER - Bright Minds Academy
=====
Devices tracked: 0

-- Add new device -- (Esc to cancel) --

Device name: Ceiling Fan E5
Room: Living Room

Device added: D001 - Ceiling Fan E5 (Living Room).

=====
SMART HOME DEVICE MANAGER - Bright Minds Academy
=====
Devices tracked: 1

-- Add new device -- (Esc to cancel) --

Device name:
Room: Living Room

Error: Device name and room cannot be empty.

=====
SMART HOME DEVICE MANAGER - Bright Minds Academy
=====
Devices tracked: 1

-- Add new device -- (Esc to cancel) --

Device name: Ceiling Fan E5
Room: Bedroom

Error: Device 'Ceiling Fan E5' already exists.
```

Menu-driven evidence for add_device()

The automated checks confirm two devices can be added with sequential ids (D001, D002), that a duplicate name gets rejected, and that an empty or whitespace-only name gets rejected. The screen capture on the right shows the same rules from the menu: 'Ceiling Fan E5' is accepted and assigned an id, a blank name is rejected with the exact validation message, and using the name a second time is rejected as a duplicate.

update_status() / update_device()

```
test_smart_device_manager.py - demo() : update_status / update_device checks

updated = m.update_status(d1.id, "online")
assert updated.status == "online"

try:
    m.update_status(d1.id, "flying")
    assert False, "bad status should have raised"
except ValueError:
    pass

try:
    m.update_status("D999", "online")
    assert False, "unknown id should have raised"
except ValueError:
    pass

updated = m.update_device(d1.id, new_name="", new_room="Bedroom", new_status="")
assert updated.id == "D001"
assert updated.room == "Bedroom"
assert updated.name == "Smart Light A1"
assert updated.status == "online"

try:
    m.update_device(d1.id, new_name="Projector B2")
    assert False, "renaming to a duplicate name should have raised"
except ValueError:
    pass
```

```
python main.py -- Update device status

=====
SMART HOME DEVICE MANAGER - Bright Minds Academy
=====
Devices tracked: 5

-- Update device --

ID      | Name           | Room           | Status           | Last Used
-----|-----|-----|-----|-----
D001    | Ceiling Fan E5 | Living Room    | offline          | 2026-08-10 16:17:10
D002    | Smart Light A1 | Living Room    | online           | 2026-08-10 16:17:10
D003    | Projector B2   | Classroom 1    | offline          | 2026-08-10 16:17:10
D004    | Thermostat C3  | Office         | under maintenance | 2026-08-10 16:17:10
D005    | Smart Light D4 | Classroom 2    | online           | 2026-08-10 16:17:10

Device id to update (Esc to cancel): D001

Current: D001 | Ceiling Fan E5 | Living Room | offline | 2026-08-10 16:17:10
Id: D001 (cannot be changed)
Leave a field blank to keep its current value. Press Esc to cancel.
New name [Ceiling Fan E5]:
New room [Living Room]:
Valid statuses: online, offline, under maintenance
New status [offline]: online

Updated: D001 | Ceiling Fan E5 | Living Room | online | 2026-08-10 16:17:10

=====
SMART HOME DEVICE MANAGER - Bright Minds Academy
=====
Devices tracked: 5

-- Update device --

Device id to update (Esc to cancel): D001
New status [online]: flying

Error: Status must be one of ('online', 'offline', 'under maintenance').

=====
SMART HOME DEVICE MANAGER - Bright Minds Academy
=====
Devices tracked: 5

-- Update device --

Device id to update (Esc to cancel): D999

Error: Device id 'D999' not found.
```

Menu-driven evidence for update_status() / update_device()

The automated checks confirm a valid status change gets applied, an invalid status string ('flying') gets rejected, an unknown id gets rejected, a partial update leaves blank fields unchanged while the id stays fixed, and renaming to a name already in use gets rejected. The screen capture shows the equivalent menu flow: a device moved from offline to online successfully, 'flying' rejected with the list of valid statuses, and an unknown id rejected before any update logic runs.

search()

```
test_smart_device_manager.py - demo() : search checks

assert m.search("d001")[0].id == "D001"
assert any(d.name == "Projector B2" for d in m.search("projector"))
assert any(d.room == "Bedroom" for d in m.search("bedroom"))
assert any(d.status == "online" for d in m.search("online"))

try:
    m.search("Nope")
    assert False, "missing device search should have raised"
except ValueError:
    pass
```

```
python main.py -- Advanced search

=====
SMART HOME DEVICE MANAGER - Bright Minds Academy
=====
Devices tracked: 5

-- Advanced search -- (Esc to cancel) --

Search by id, name, room, or status: light

2 match(es) found:

ID      | Name           | Room           | Status  | Last Used
-----|-----|-----|-----|-----
D002    | Smart Light A1 | Living Room    | online  | 2026-08-10 16:17:10
D005    | Smart Light D4 | Classroom 2    | online  | 2026-08-10 16:17:10

=====
SMART HOME DEVICE MANAGER - Bright Minds Academy
=====
Devices tracked: 5

-- Advanced search -- (Esc to cancel)--

Search by id, name, room, or status: xyz999

Error: No device matches 'xyz999'.
```

Menu-driven evidence for search()

The automated checks confirm a query matches by id, name, room and status (case-insensitive substring), and that a query with no matches gets rejected rather than silently returning nothing. The screen capture shows 'light' correctly matching two devices, and 'xyz999' correctly rejected as no match.

delete_device()

```
test_smart_device_manager.py - demo() : delete_device checks
```

```
try:
    m.delete_device("D999")
    assert False, "deleting unknown id should have raised"
except ValueError:
    pass

removed = m.delete_device(d2.id)
assert removed.id == "D002"
assert len(m.devices) == 1
assert len(m.deleted) == 1
try:
    m.get_by_id("D002")
    assert False, "deleted device should no longer resolve as active"
except ValueError:
    pass

try:
    m.search("projector")
    assert False, "deleted device should not appear in search"
except ValueError:
    pass
```

```
python main.py -- Delete + Recover device

=====
SMART HOME DEVICE MANAGER - Bright Minds Academy
=====
Devices tracked: 5

-- Delete device -- (Esc to cancel) --

ID      | Name           | Room           | Status           | Last Used
-----|-----|-----|-----|-----
D001    | Ceiling Fan E5 | Living Room    | online           | 2026-08-10 16:17:10
D002    | Smart Light A1 | Living Room    | online           | 2026-08-10 16:17:10
D003    | Projector B2   | Classroom 1    | offline          | 2026-08-10 16:17:10
D004    | Thermostat C3  | Office         | under maintenance | 2026-08-10 16:17:10
D005    | Smart Light D4 | Classroom 2    | online           | 2026-08-10 16:17:10

Device id to delete: D001

About to delete: D001 | Ceiling Fan E5 | Living Room | online | 2026-08-10
Are you sure you want to delete this device? (y/n): y
Type the device id 'D001' to confirm: D001

Deleted: D001 - Ceiling Fan E5. (recoverable via 'Recover deleted device')

=====
SMART HOME DEVICE MANAGER - Bright Minds Academy
=====
Devices tracked: 4

-- Recover deleted device -- (Esc to cancel) --

ID      | Name           | Room           | Status           | Last Used
-----|-----|-----|-----|-----
D001    | Ceiling Fan E5 | Living Room    | online           | 2026-08-10 16:17:10

Device id to recover: D999

Error: Deleted device id 'D999' not found.

=====
SMART HOME DEVICE MANAGER - Bright Minds Academy
=====
Devices tracked: 4

-- Recover deleted device -- (Esc to cancel) --

ID      | Name           | Room           | Status           | Last Used
-----|-----|-----|-----|-----
D001    | Ceiling Fan E5 | Living Room    | online           | 2026-08-10 16:17:10

Device id to recover: D001

Recovered: D001 - Ceiling Fan E5.
```

Menu-driven evidence for delete_device()

The automated checks confirm deleting an unknown id gets rejected, a valid delete moves the device out of the active list into the recoverable list, and the deleted device no longer resolves via get_by_id() or shows up in search results. The screen capture shows the real double-confirmation delete flow (y/n, then retype the id) succeeding.

recover_device()

```
test_smart_device_manager.py - demo() : recover_device checks

try:
    m.recover_device("D999")
    assert False, "recovering unknown id should have raised"
except ValueError:
    pass

recovered = m.recover_device("D002")
assert recovered.id == "D002"
assert len(m.devices) == 2
assert len(m.deleted) == 0
assert m.get_by_id("D002").name == "Projector B2"

m.delete_device("D002")
m.add_device("Projector B2", "Room 9")
try:
    m.recover_device("D002")
    assert False, "recovering into a name collision should have raised"
except ValueError:
    pass

print("All checks passed.")
```

```
python main.py -- Delete + Recover device

=====
SMART HOME DEVICE MANAGER - Bright Minds Academy
=====
Devices tracked: 5

-- Delete device -- (Esc to cancel) --

ID | Name | Room | Status | Last Used
-----
D001 | Ceiling Fan E5 | Living Room | online | 2026-08-10 16:17:10
D002 | Smart Light A1 | Living Room | online | 2026-08-10 16:17:10
D003 | Projector B2 | Classroom 1 | offline | 2026-08-10 16:17:10
D004 | Thermostat C3 | Office | under maintenance | 2026-08-10 16:17:10
D005 | Smart Light D4 | Classroom 2 | online | 2026-08-10 16:17:10

Device id to delete: D001

About to delete: D001 | Ceiling Fan E5 | Living Room | online | 2026-08-10
Are you sure you want to delete this device? (y/n): y
Type the device id 'D001' to confirm: D001

Deleted: D001 - Ceiling Fan E5. (recoverable via 'Recover deleted device')

=====
SMART HOME DEVICE MANAGER - Bright Minds Academy
=====
Devices tracked: 4

-- Recover deleted device -- (Esc to cancel) --

ID | Name | Room | Status | Last Used
-----
D001 | Ceiling Fan E5 | Living Room | online | 2026-08-10 16:17:10

Device id to recover: D999

Error: Deleted device id 'D999' not found.

=====
SMART HOME DEVICE MANAGER - Bright Minds Academy
=====
Devices tracked: 4

-- Recover deleted device -- (Esc to cancel) --

ID | Name | Room | Status | Last Used
-----
D001 | Ceiling Fan E5 | Living Room | online | 2026-08-10 16:17:10

Device id to recover: D001

Recovered: D001 - Ceiling Fan E5.
```

Menu-driven evidence for recover_device()

The automated checks confirm recovering an unknown id gets rejected, a valid recovery restores the device to the active list, and, the trickiest edge case, recovering is blocked if an active device has since taken the recovered device's old name. The same screen capture as above shows an unknown-id recovery being rejected and a valid recovery succeeding.

Full Suite Run

```
python test_smart_device_manager.py

All checks passed.
```

Figure K: test_smart_device_manager.py executing end to end. Every assertion above ran in one pass, and the script only prints this success line if all of them passed.

Table 1 lists every individual assertion in test_smart_device_manager.py against the DeviceManager method it targets, the input used, the expected result, and the actual outcome observed when the suite last ran. All 24 passed.

Table 1: Test case matrix for test_smart_device_manager.py.

Test ID	Method	Input	Expected Result	Actual Outcome	Result
T01	add_device	name='Smart Light A1', room='Room 1'	Device created, id = D001	id == 'D001'	PASS
T02	add_device	name='Projector B2', room='Room 2'	Device created, id = D002	id == 'D002'	PASS
T03	add_device	name='Smart Light A1' (duplicate)	ValueError raised	ValueError raised	PASS
T04	add_device	name=' ' (blank)	ValueError raised	ValueError raised	PASS
T05	update_status	d1.id, 'online'	status == 'online'	status == 'online'	PASS
T06	update_status	d1.id, 'flying'	ValueError raised	ValueError raised	PASS
T07	update_status	'D999', 'online'	ValueError raised	ValueError raised	PASS
T08	update_device	room='Bedroom', name/status blank	Only room changes; id unchanged	room == 'Bedroom', id == 'D001'	PASS
T09	update_device	new_name='Projector B2' (taken)	ValueError raised	ValueError raised	PASS
T10	search	'd001'	Matches D001	result[0].id == 'D001'	PASS
T11	search	'projector'	Matches Projector B2	match found	PASS
T12	search	'bedroom'	Matches by room	match found	PASS
T13	search	'online'	Matches by status	match found	PASS
T14	search	'Nope'	ValueError raised	ValueError raised	PASS
T15	delete_device	'D999'	ValueError raised	ValueError raised	PASS
T16	delete_device	d2.id	Moved to deleted list	len(devices)=1, len(deleted)=1	PASS
T17	get_by_id	'D002' (after delete)	ValueError raised	ValueError raised	PASS
T18	search	'projector' (after delete)	ValueError raised (excluded)	ValueError raised	PASS
T19	recover_device	'D999'	ValueError raised	ValueError raised	PASS
T20	recover_device	'D002'	Restored to active list	len(devices)=2, len(deleted)=0	PASS
T21	get_by_id	'D002' (after recover)	name == 'Projector B2'	name == 'Projector B2'	PASS
T22	delete_device + add_device	delete D002, add 'Projector B2'	New device reuses freed name	add succeeds	PASS
T23	recover_device	'D002' (name now reused)	ValueError raised	ValueError raised	PASS
T24	(suite)	-	'All checks passed.' printed	printed	PASS

4. Evaluation

4.1. Challenges Faced:

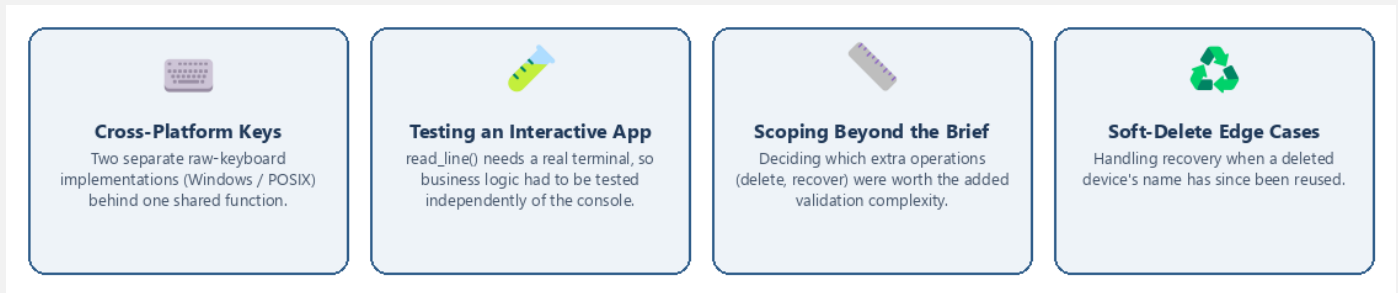


Figure L: The four main challenges encountered during development.

- ❖ Cross-platform keyboard input: Python's built-in `input()` only returns after Enter, so it cannot see an Esc keypress at all. Solving that properly meant dropping down to raw, per-keystroke console reading, writing and testing two separate implementations (`msvcrt` on Windows, `termios/tty` on Linux/macOS), and keeping them behind one shared `read_line()` interface so the rest of the program never has to know which platform it is running on.
- ❖ Testing an interactive console app: that same raw-keyboard requirement makes the UI hard to test automatically, since a script cannot simply pipe text into it the way it could with `input()`. The solution used here (Section 3.3) was to test the validated business logic in `DeviceManager` directly and treat the console layer as a thin shell verified separately.
- ❖ Scoping beyond the brief's four minimum operations: delete and recover got added because leaving no way to remove a mistaken entry felt like a genuine gap for real staff to hit, but each added operation also brings edge cases that then need thinking through and validating, so scope had to be weighed carefully rather than added by default.
- ❖ Soft-delete edge cases: specifically, what should happen if a deleted device's name gets reused by a new device before the original is recovered. This one was only caught because the design was sketched out on paper (Section 2.1) before it was coded.

4.2 Future Improvements:

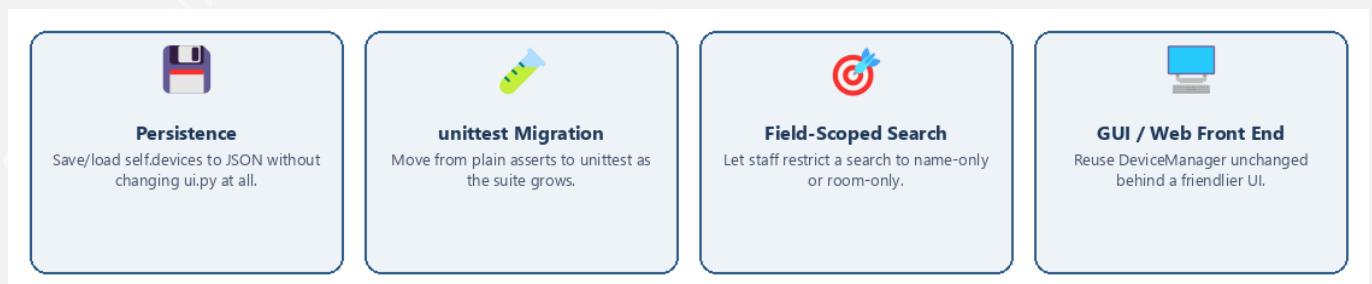


Figure M: The four highest-value improvements for a future version.

- ❖ Persistence: the brief rules out a database for this version, but the layered architecture in Section 2.2 was chosen partly so that adding save/load methods to `DeviceManager` (for example, writing

self.devices to a JSON file on exit and reading it back on start-up) would not need touching ui.py at all.

- ❖ Migrate to unittest: moving the automated tests from plain assert statements to Python's unittest module would give clearer per-test pass/fail reporting as the suite grows beyond what one demo() function can comfortably cover [12].
- ❖ Field-scoped search: the current 'search matches any field' behaviour is convenient but occasionally too broad. A future version could let staff optionally restrict a search to a single field, name only, or room only.
- ❖ A GUI or web front end: reusing DeviceManager completely unchanged, since it has no dependency on the console at all, would open the system up to staff who are less comfortable with a keyboard-driven menu, without discarding any validation logic already built and tested.

4.3 Self-Reflections:



Figure N: What this project contributed to my own growth as a developer.

Conclusion

This report has covered the analysis, design, implementation, testing and evaluation of a Smart Home Device Manager built for Bright Minds Academy's smart-classroom pilot. Working from the four minimum operations set out in the brief, requirements analysis identified three further operations, delete, recover, and multi-field search, that a genuine classroom deployment would need, and these got implemented alongside the originals using an object-oriented design in which a Device class models a single device and a DeviceManager class owns every validated operation performed on the collection of devices. The system was designed on paper first, using flowcharts and pseudo-code, before being implemented across five focused Python modules using only the standard library, and it was tested both through an automated script exercising DeviceManager's validation logic directly and through a scripted walkthrough of the actual menu-driven screens, covering valid and invalid input for every operation. The resulting program answers the problem set out in the introduction: staff can now add, update, view, search for, delete and recover smart devices through a single, fast, validated console interface, without needing a database or any training beyond reading an on-screen menu.

References

- [1] Python Software Foundation, "9. Classes," Python 3 Documentation, 2024. [Online]. Available: <https://docs.python.org/3/tutorial/classes.html>
- [2] Python Software Foundation, "datetime - Basic date and time types," Python 3 Documentation, 2024. [Online]. Available: <https://docs.python.org/3/library/datetime.html>
- [3] Python Software Foundation, "Errors and Exceptions," Python 3 Documentation, 2024. [Online]. Available: <https://docs.python.org/3/tutorial/errors.html>
- [4] G. van Rossum, B. Warsaw, and N. Coghlan, "PEP 8 - Style Guide for Python Code," Python Enhancement Proposals, 2001. [Online]. Available: <https://peps.python.org/pep-0008/>
- [5] I. Sommerville, Software Engineering, 10th ed. Harlow, U.K.: Pearson Education, 2016.
- [6] R. S. Pressman and B. R. Maxim, Software Engineering: A Practitioner's Approach, 9th ed. New York, NY, USA: McGraw-Hill Education, 2020.
- [7] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship. Upper Saddle River, NJ, USA: Prentice Hall, 2008.
- [8] G. J. Myers, C. Sandler, and T. Badgett, The Art of Software Testing, 3rd ed. Hoboken, NJ, USA: Wiley, 2011.
- [9] IEEE, "IEEE Standard for Software and System Test Documentation," IEEE Std 829-2008, 2008.
- [10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA, USA: Addison-Wesley, 1994.
- [11] K. Beck et al., "Manifesto for Agile Software Development," 2001. [Online]. Available: <https://agilemanifesto.org/>
- [12] Python Software Foundation, "unittest - Unit testing framework," Python 3 Documentation, 2024. [Online]. Available: <https://docs.python.org/3/library/unittest.html>

Appendices

Appendix A: Automated Test Coverage Map

Each row below maps a block inside `test_smart_device_manager.py` (Section 2.3) to the requirement or edge case it verifies; this supplements the screenshots in Section 3.3 with a full list of what the automated layer checks on every run.

Scenario tested	Requirement / edge case verified
Add device with valid name and room	Functional requirement: add new device; auto id assignment (D001, D002, ...)
Add device with a duplicate name	Validation: duplicate-name rejection (<code>device_manager.py, _find_by_name</code>)
Add device with a blank/whitespace-only name	Validation: empty name/room rejection
Update status to a valid value	Functional requirement: update device status; <code>last_used</code> timestamp refresh
Update status to an invalid value ('flying')	Validation: status must be one of <code>VALID_STATUSES</code>
Update status for an unknown device id	Validation: unknown-id rejection
Partial update (room only, name/status blank)	Business rule: blank fields keep their current value; id is immutable
Rename a device to a name already in use	Validation: duplicate-name rejection also applies on rename
Search by id, name, room, and status	Functional requirement: advanced multi-field search
Search with no matching device	Validation: 'no match' rejection instead of a silent empty result
Delete an unknown device id	Validation: unknown-id rejection on delete
Delete a valid device	Functional requirement: soft delete moves the device to the recoverable list

Search after deletion	Business rule: deleted devices are excluded from active search results
Recover an unknown deleted device id	Validation: unknown-id rejection on recover
Recover a valid deleted device	Functional requirement: recover restores the device to the active list
Recover into a reused name	Business rule: recovery is blocked if an active device now holds that name

Appendix B: Quick Reference, Main Menu Options

Reproduced from ui.py's print_menu() for quick reference alongside the screenshots in Section 3.3.

Option	Screen
1	Add new device
2	Update device (name, room, or status)
3	View all devices
4	Advanced search (by id, name, room, or status)
5	Delete device
6	Recover deleted device
7	Exit