Ashton Winnett

Student ID: 1184199

Assessment Task 2

ICT112: Introduction to Programing Fundamentals, May 2025  
Tutor: Kylie Docherty

**Flood Emergency Management System Report**

Contents

[Introduction 2](#_Toc197693055)

[Core Functionality 2](#_Toc197693056)

[Scope, Limitations, and Assumptions 3](#_Toc197693057)

[Design Approach 3](#_Toc197693058)

[Initial Plan 4](#_Toc197693059)

[Final Plan 5](#_Toc197693060)

[Libraries Used 8](#_Toc197693061)

[Testing the Program 8](#_Toc197693062)

[Automated Tests 8](#_Toc197693063)

[Manual Tests 9](#_Toc197693064)

[References 10](#_Toc197693065)

# Introduction

This report presents an overview of the Flood Emergency Management System, a Python-based project developed from a foundational text adventure game provided in class. The system has been expanded into a comprehensive tool designed to manage flood crises across a region comprising multiple suburbs. Key features include navigation between locations, resource management (e.g., sandbags), monitoring of evacuation centres, and dynamic handling of fluctuating flood levels. The codebase incorporates automated testing and clear, well-documented comments to ensure maintainability and readability.

The report is structured to detail the system’s functionality, design process, and testing methodology. It also outlines the assumptions made during development and the limitations of the task.

# Core Functionality

The Flood Emergency Management System incorporates the following key features:

**Navigation**: Movement between suburbs is facilitated, with access restricted to locations where flood levels reach or exceed 10.

**Resource Management**: Resources, such as sandbags, can be collected and distributed to areas in need.

**Evacuation Centres**: Each suburb may contain evacuation centres, with detailed information on available services and accommodations.

**Data Persistence**: The system’s state is preserved in a JSON file named suburbs.json.

**Dynamic Flood Levels**: Flood levels adjust incrementally each turn, introducing increased complexity to navigation and resource allocation.

**Command Processing**: A dedicated class validates and processes user inputs to ensure accurate command execution.

**Automated Testing**: Comprehensive automated tests are integrated to verify system functionality.

**Code Clarity**: Extensive comments are embedded throughout the codebase to enhance readability and maintainability.

# Scope, Limitations, and Assumptions

The Flood Emergency Management System is constrained to four suburbs to ensure manageability, though its JSON-based structure supports the addition of more suburbs as needed, provided sufficient computer memory is available. A login system was excluded, as it was not required for the project. The flood mechanics are simplified, relying on random level changes rather than complex real-world weather or flood pattern modelling. To facilitate input parsing, items are represented as single words, such as “sandbags.” User inputs are expected to follow straightforward commands like “go north” or “take sandbags,” with the program providing error feedback for invalid entries. The system is designed to revert to default data if the JSON file becomes corrupted, preventing crashes. It is assumed that the computer has adequate storage for saving the JSON file and that no network issues interfere with file operations. The random flood level adjustments enhance gameplay variability without attempting to replicate intricate real-world flood dynamics. These assumptions and limitations enabled the development of a practical and functional system while addressing the problem’s complexities effectively.

Design Approach

The Flood Emergency Management System was developed using Python classes to ensure a structured and organised codebase, with distinct classes for managing suburbs, the player, and overall system operations. Functionality for data persistence, dynamic flood level adjustments, and automated testing was integrated to enhance reliability and maintainability. Extensive comments were incorporated throughout the code to ensure clarity and accessibility for all users.

## Initial Plan

The development process for the Flood Emergency Management System began with a conceptual framework focused on creating an effective solution. This involved outlining the core structure and functionality, using pseudocode, to ensure the system would meet its objectives efficiently. The initial design is outlined as follows:

```

Start with empty suburbs, a player, and set Central as the starting point

Load suburb data from a file, or use defaults if there’s no file:

    Central: has sandbags, radio, an evacuation centre, and links to other suburbs

    Riverside: has a water pump, no centres, links

    Hilltop: has sandbags, a centre, links

    Parkside: has a first aid kit, a centre, links

While there are turns left:

    Show the suburb’s details, items, and centres

    Ask for a command

    If they type “quit”:

        Save everything

        Stop

    If they type “go” and a direction:

        If the direction works, move there

        Else, show an error

    If they type “take” and an item:

        If the item’s there, pick it up

        Else, show an error

    If they type “drop” and an item:

        If they’re carrying it, drop it

        Else, show an error

    If they type “inventory”:

        Show what they’re carrying

    Reduce turns by 1

Save everything

Say the game’s over

```

## Final Plan

Following development, the Flood Emergency Management System was enhanced to incorporate advanced features and refined functionality. The finalised design is outlined as follows:

```

CLASS CommandParser:

    Has a list of okay commands

    METHOD parse(input):

        If input’s empty, say so

        Split into command and args

        If command’s not okay, show error

        Return command and args

CLASS Suburb:

    Has name, description, items, centres, connections, flood level

    METHOD describe:

        Show all details, including flood level

CLASS Admin:

    Has name and inventory

    METHOD take(item, suburb):

        If item’s there, pick it up and show what’s carried

        Else, show error

    METHOD drop(item, suburb):

        If carrying item, drop it and show where

        Else, show error

    METHOD show\_inventory:

        Show what’s carried

CLASS EmergencyManager:

    Has suburbs, current suburb, admin, file name, parser

    METHOD load\_suburbs:

        Try to load JSON file

        If it’s broken, use defaults and save them

        Make Suburb objects

        Start in Central

    METHOD save\_suburbs:

        Try to save data

        If it fails, show error

    METHOD update\_flood\_levels:

        Randomly tweak flood levels for each suburb

    METHOD move(direction):

        If direction’s valid and flood’s not too bad, move

        Else, show error

    METHOD main:

        Get admin’s name

        For 50 turns:

            Update flood levels

            Show suburb

            Parse command

            If error, show it

            If quit, save and stop

            If go, try to move

            If take, try to pick up

            If drop, try to drop

            If inventory, show it

        Save data

        Say it’s done

CLASS TestEmergencyManager:

    METHOD setUp:

        Make a temp file for testing

        Start a test game

    METHOD tearDown:

        Delete temp file

    Tests for:

        Starting up, moving, flood blocks, taking, dropping, inventory, saving, JSON errors

If program run with “--test”:

    Run tests

Else:

    Run game

```

## Libraries Used

The Flood Emergency Management System uses several Python libraries to ensure robust functionality and efficient operation. The *json* library facilitates the saving and loading of suburb data, with error handling for corrupted files. The *os* library verifies the existence of the JSON file, while the *sys* library detects the “--test” flag to initiate automated testing. The *unittest* library executes comprehensive automated tests, and the *tempfile* library creates temporary files for testing to safeguard actual data. The *shutil* library manages the cleanup of test files, and the *random* library enables dynamic, randomised changes to flood levels. All functionality is supported using standard Python 3.9, requiring no additional downloads.

# Testing the Program

The Flood Emergency Management System was evaluated through two complementary testing approaches: automated and manual. Automated tests, executed independently, utilise a temporary file to avoid modifying the primary *suburbs.json* file, ensuring data integrity. These tests can be initiated using the command python *flood\_emergency\_manager.py --test*. Manual testing involved interactive sessions to verify system functionality, confirming that all features operate as intended across various scenarios.

## Automated Tests

The automated tests for the Flood Emergency Management System are designed to evaluate the core components of the program independently, requiring no manual intervention. These tests systematically verify the functionality of key features, ensuring reliable performance across the system.

| **Test ID** | **What It Checks** | **Test Name** | **What Should Happen** |

| AT01 | Does it start right? | `test\_initialisation` | Starts in Central, 4 suburbs, sandbags there |

| AT02 | Can you move okay? | `test\_move\_valid` | Goes to Riverside, says so |

| AT03 | What if you can’t move? | `test\_move\_invalid` | Stays in Central, shows error |

| AT04 | Flood stops you? | `test\_move\_flood\_blocked` | Can’t go to Riverside if flooded, shows error |

| AT05 | Pick up stuff | `test\_take\_item` | Gets sandbags, updates lists |

| AT06 | Pick up wrong stuff | `test\_take\_invalid\_item` | Nothing happens, shows error |

| AT07 | Drop stuff | `test\_drop\_item` | Drops sandbags, updates lists |

| AT08 | Drop wrong stuff | `test\_drop\_invalid\_item` | Nothing happens, shows error |

| AT09 | Show inventory | `test\_inventory` | Shows “sandbags, radio” |

| AT10 | Save and load data | `test\_data\_persistence` | Sandbags stay gone after reload |

| AT11 | Bad JSON file | `test\_json\_error\_handling` | Uses defaults if file’s broken |

## Manual Tests

In addition to automated testing, the Flood Emergency Management System was manually tested through interactive gameplay sessions. This approach ensured that the system operates and delivers a smooth user experience, confirming that all features function as intended in a real-world context.

| **Test ID** | **What I Did** | **Input** | **What Should Happen** | **What Happened** |

| MT01 | Started it | Entered “Ashton” | Shows Central, flood level 0 | Worked fine |

| MT02 | Moved | `go north` | Goes to Riverside, shows details | As expected |

| MT03 | Tried bad move | `go south` | Says you can’t | Good |

| MT04 | Picked up | `take sandbags` | Says I got sandbags | Perfect |

| MT05 | Picked up wrong | `take boat` | Says it’s not there | Correct |

| MT06 | Dropped | `drop sandbags` | Says dropped in Riverside | All good |

| MT07 | Dropped wrong | `drop boat` | Says I’m not carrying it | Fine |

| MT08 | Checked empty inventory | `inventory` | Says “none” | Worked |

| MT09 | Checked full inventory | `inventory` | Shows “sandbags” | As expected |

| MT10 | Quit | `quit` | Says it’s exiting | Good |

| MT11 | Bad command | `hello` | Says it’s unknown | Correct |

| MT12 | No input | Pressed Enter | Says to enter a command | Fine |

| MT13 | Ran tests | `python ... --test` | All tests pass | Passed |

| MT14 | Flood stopped me | `go north` with flood | Says flooding blocks it | Worked |

# References

The Flood Emergency Management System was developed using only the foundational code created in class and Python’s official help files. The system’s design and implementation were taken from classroom learning and original ideas, ensuring a tailored approach to the project requirements.

**AI Use and Acknowledgment**

The development of the Flood Emergency Management System incorporated Grok, an AI assistant created by xAI, to address targeted coding challenges. Grok was used in troubleshooting and resolving code-related issues, including debugging errors, optimising logic, and refining implementation strategies. For instance, when faced with functionality problems, Grok provided actionable suggestions to diagnose errors and propose effective solutions, thereby enhancing the efficiency of the development process. Grok’s role in code troubleshooting and optimisation is acknowledged. All other project elements, including conceptualisation, design, and core implementation, were independently developed using only class materials and original contributions.