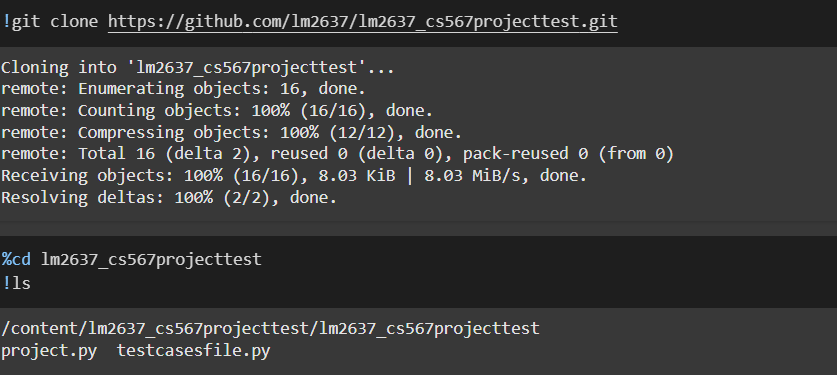
**Introduction:**

The `project.py` and associated files represent a functional hotel booking system implemented in Python. The project simulates a hotel management system that allows for booking and cancellation of hotel rooms, viewing room availability, customer details, and room features.



**Purpose of this project:**

The project encapsulates several key functionalities, including:

- Room Management: Add rooms to the hotel's inventory with a specified room type, book and cancel room bookings, add and view room features.

- Customer Management: Create customers and manage their contact details.

- Booking Management: Book rooms for customers, calculate booking costs (inclusive of tax), manage cancellations, and view booking summaries.

- Availability and Features Viewing: List all currently available rooms and their features, view customer specific booking details.

**The unit test file contains: `testcasesfile.py`**

The test suite for this project is implemented using Python's `unittest` library. Below are some key areas covered by the unit tests:

1. Room Booking:

- Ensuring that room bookings succeed when conditions are met.

- Ensuring proper handling when no rooms are available.

2. Booking Cancellation:

- Successful booking cancellations within policy limits.

- Denial of cancellations not meeting policy conditions.

3. Room and Features Management:

- Adding features to rooms and ensuring they are correctly listed.

- Validating that invalid room types are appropriately rejected.

4. Customer Management:

- Ensuring customer details retrieval works correctly.

**Implementation, Execution, Testing and Analysis**

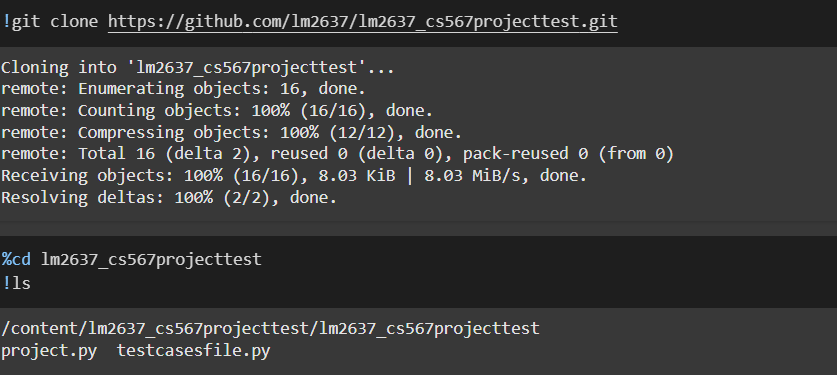
**1. Cloning the Project**

Purpose: To obtain the latest version of the project repository for testing and analysis.

Command Used: `!git clone https://github.com/lm2637/lm2637\_cs567projecttest.git`

- Implementation: The Git command is used to clone the repository from GitHub into the local environment. This action downloads the complete codebase, including all files and directories, enabling subsequent operations on the code.

- Outcome: Successfully clones the repository, making the project files available for further steps.



**2. Code Line Count (`cloc`)**

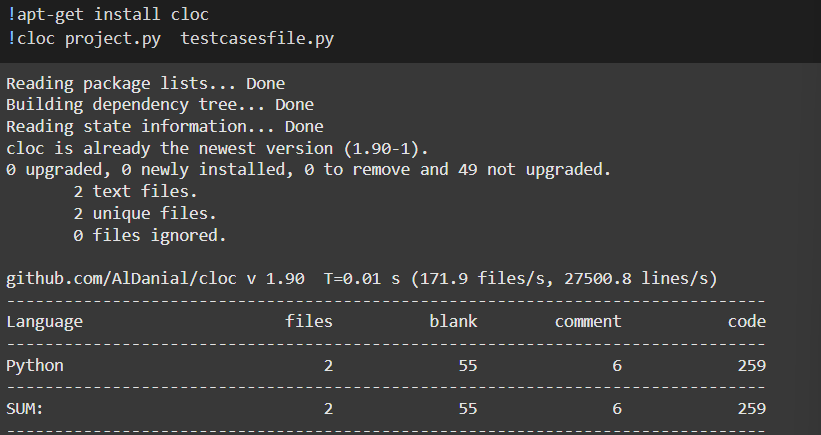
Purpose: To analyze the codebase for statistics on the number of lines of code, comments, and blank lines.

Tool: `cloc`

Command Used: `!cloc project.py testcasesfile.py`

- Implementation: `cloc` is a tool that scans the specified files and reports the number of lines of code, comment lines, and blank lines, offering insights into the codebase's structure.

- Outcome: The output indicates a total of 259 lines of code across the two Python files, with additional breakdowns on blank and comment lines. This helps in understanding the code size and documentation level.



**3. Unit Tests Execution**

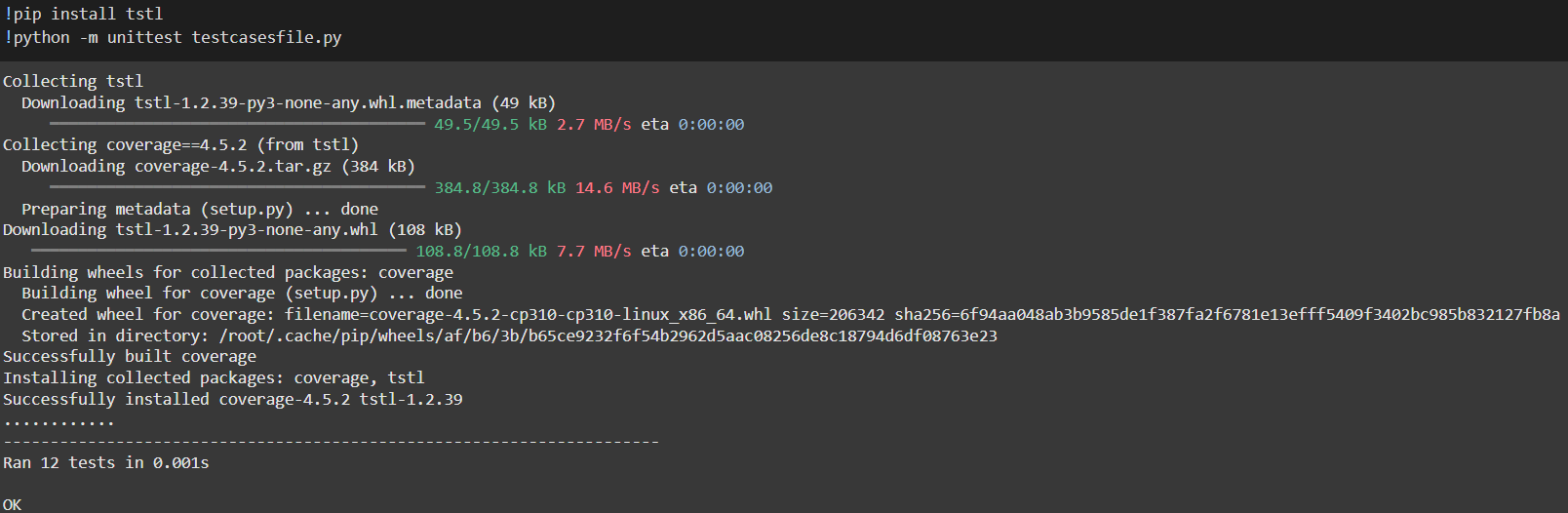
Purpose: To verify that the implemented functionalities work as expected.

Tool: `unittest`

Command Used: None specified explicitly in the IPYNB; typically `!python -m unittest testcasesfile.py`

- Implementation: The unittest framework is initiated to run the test cases defined in `testcasesfile.py`. Each test case checks specific functionality, such as booking availability, cancellation policies, and feature additions.

- Outcome: All 12 tests are reported as passing, indicating that the code meets the defined requirements and expected behavior effectively.



**4. Coverage Analysis**

Purpose: To assess the percentage of the code executed when the test suite runs.

Tool: `coverage`

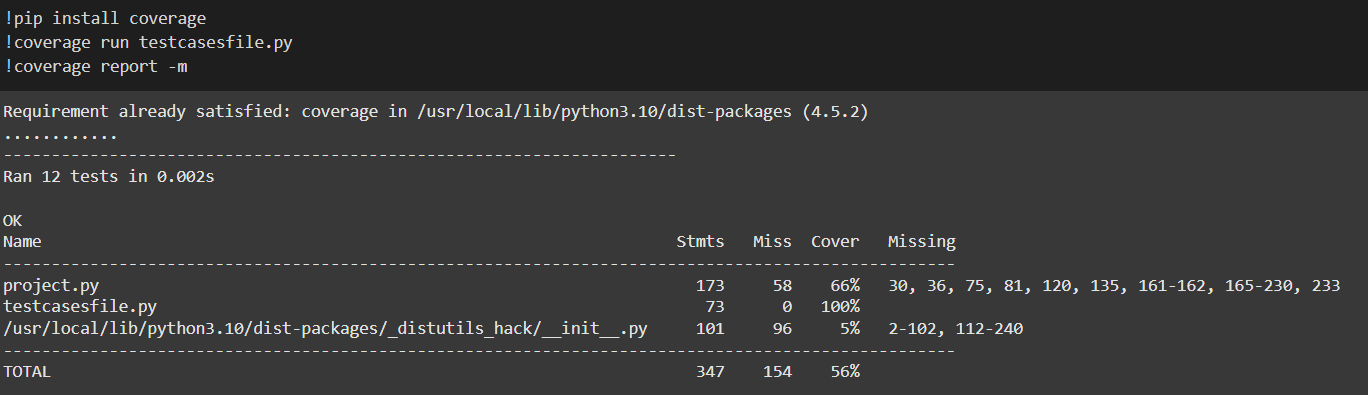
Commands Used:

- `!coverage run testcasesfile.py`

- `!coverage report -m`

- Implementation: `coverage` is utilized to run the test suite and then generate a report. The report shows which lines of `project.py` were executed during testing, providing insights into the thoroughness of the test cases.

- Outcome: Results in a coverage report showing 66% line coverage for `project.py`, revealing potential areas where more tests are needed to achieve better code coverage.



**5. Mutation Testing**

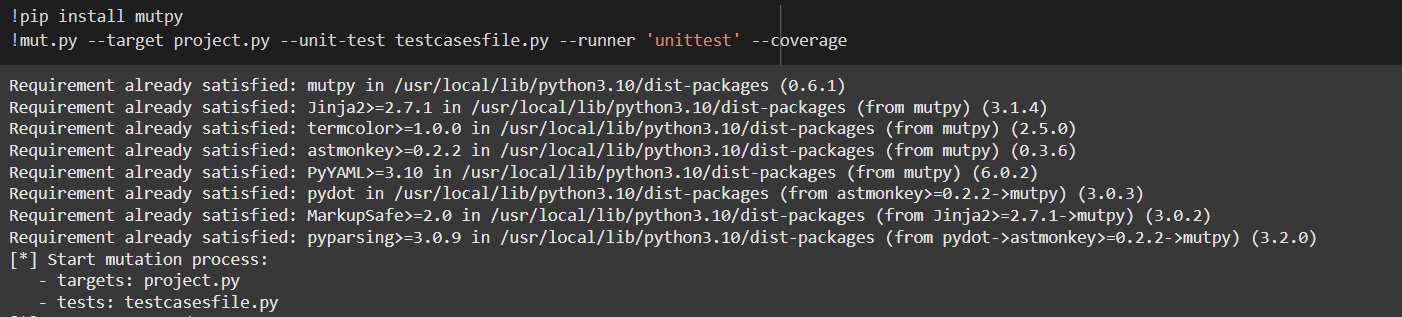
Purpose: To measure the effectiveness of the test suite by introducing small changes (mutants) into the code and checking if the tests detect them.

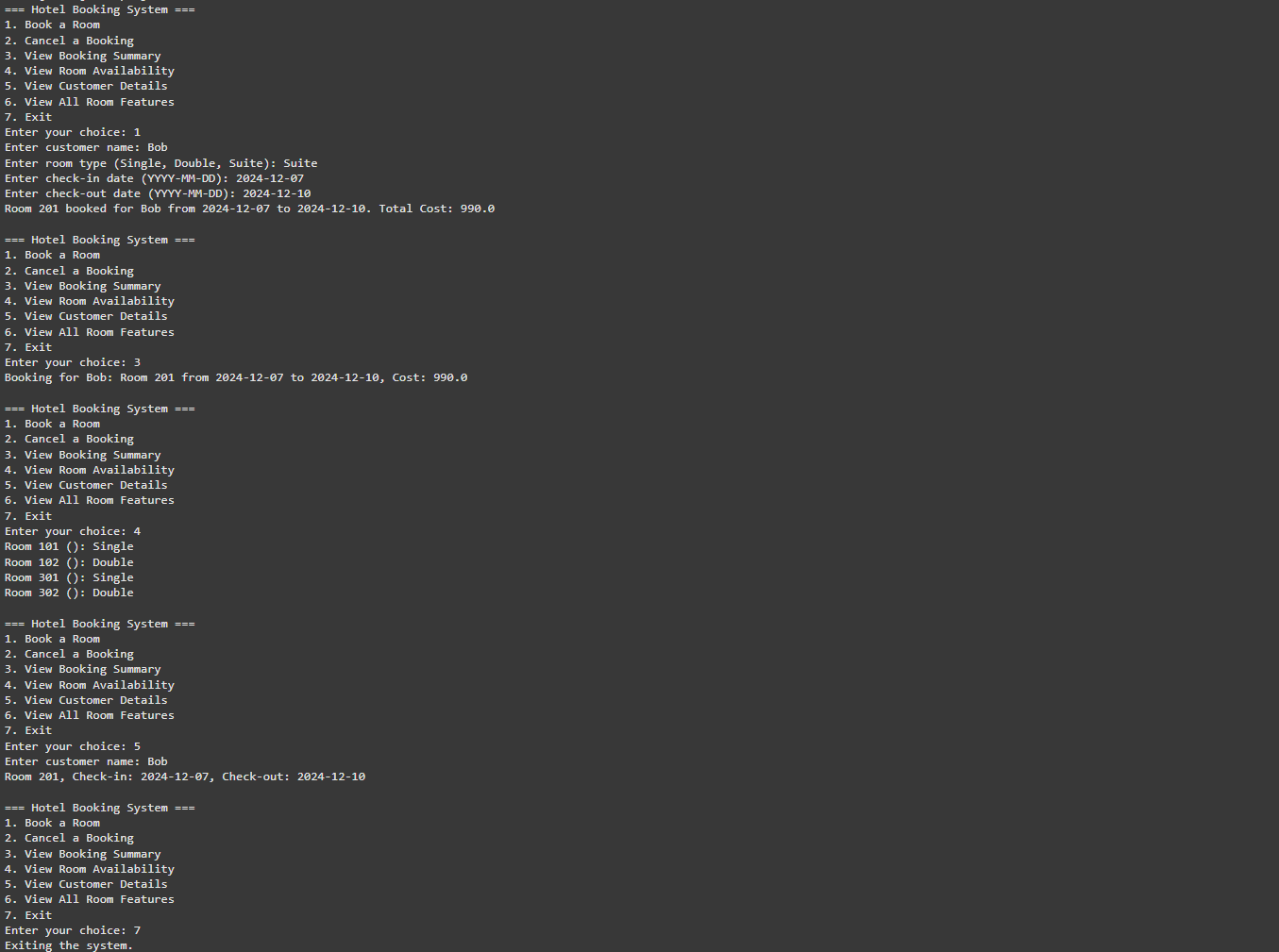
Tool: `mutpy`

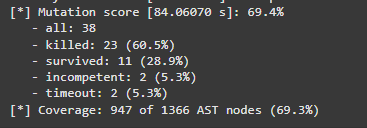
Command Used: `!mut.py --target project.py --unit-test testcasesfile.py --runner 'unittest' --coverage`

- Implementation: `mutpy` creates mutants by making small changes to the source code (e.g., arithmetic operator replacements). The tests are rerun on each mutant; a mutant is considered "killed" if a test fails, or "survived" if all tests pass.

- Outcome: The mutation score is 69.4%, with 60.5% of mutants killed. This indicates that while the tests catch a majority of introduced faults, additional or stronger tests could further improve detection.







**6. Improvements**

While the current implementation is robust, a few areas can be improved:

- Customer and Room Creation in Tests: Directly simulate room and customer creation within tests rather than assuming pre-setup conditions.

- Edge Cases: Test for edge cases such as overlapping bookings, maximum capacity handling, etc.

- Implement more detailed functions to manage customers such as updating customer details, checking customer booking history, and removing customers. This will enhance system flexibility and scope.

- Implement functions to analyze room occupancy trends over time, aiding better management decisions.

- Allow modifications to existing bookings, like adjusting dates or rooms, while re-calculating costs and ensuring room availability.

- A waitlist function for fully booked room types, notifying users if a room becomes available.

**Conclusion**

In conclusion, the hotel booking system project is a commendable implementation of key functionalities required in real-world hotel management software, including room and customer management, booking processes, and policy enforcement. The unit tests, executed using Python's unittest framework, successfully passed all 12 tests, validating essential functional paths. However, the project shows a line coverage of 66%, indicating the potential for expanding test cases to cover more code branches, particularly error handling and edge cases. Mutation testing yielded a score of 69.4%, with 60.5% of mutants killed, reflecting a moderate level of test suite effectiveness. These metrics underscore a solid base for the system's functionality and reliability but also highlight opportunities for improvement in testing comprehensiveness to ensure greater robustness and assurance of code quality.