Test Documentation

Unit Testing

The unit test framework used on this project is Pytest (http://doc.pytest.org/en/latest/). Pytest is a full featured testing tool that enables anyone to write either small or large scale unit tests.

To run unit tests for this program, use the following terminal commands:

```
cd test_data
pytest –v seat_assign_16201859_16203177_tests.py
```

Module Name: retrieve_data

Inputs

Variable	Description
engine	The database engine which is created as part of the main function
rows	An integer value showing the number of rows in the airline
cols	A string containing the columns names for each row

Outputs

Variable	Description
empty_seats	A list of tuples which show the layout of the airplane. Each tuple contains a row number, seat letter and a blank string for which a passenger name will be assigned.
empty_seats_per_row	A dictionary showing row numbers and their associated number of empty seats.
num_pas_refused	The current number of passengers refused a booking.
num_pas_split	The current number of passengers split from their booking group.

Unit Tests

Test Description	Unit test 1 is run on a cleaned airline_seating.db database. It tests the first instance of this function being run when the database is empty and the first booking is about to be assigned.
Results	Test Passed

Test Description	Unit test 2 is run on the test_airline_seating.db database. This database has already received a number of bookings so the outputs will be different to unit test 1.
Results	Test Passed

Module Name: test_find_row_with_n_empty_seats

Inputs

Variable	**Description**
Empty_seats_per_row	A dictionary showing row numbers and their associated number of empty seats.
Number_of_pas	Number of passages to be booked
e	A constant number ranging between 1 and the width of the airplane

Outputs

Variable	**Description**
Boolean	True or False
Key	The row number returned if a suitable row matching the number of passengers plus the constant 3 is found. If no value is found, zero is returned

Unit Tests

The number of passengers to be booked for all unit tests is three.

Test Description	UT1 looks through a completely empty airline (i.e. all rows have a dictionary value of 4) and attempts to find a row when the corresponding e constant value is 2.
Expected Output	(False, 0) – As no rows containing 5 (number of passengers (3) plus e (2) are available.
Results	Test Passed
Test Description	UT2 again looks through a completely empty airline (i.e. all rows have a dictionary value of 4) and attempts to find a row when the corresponding e constant value is 1.
Expected Output	(True, 1)

Results	Test Passed
Test Description	UT3 looks through a partially empty airline and attempts to find a row when the corresponding e constant value is 1.
Expected Output	(True, 5)
Results	Test Passed

Module Name: find_allocation_order

Inputs

Variable	**Description**
Number_of_pas	Number of passages to be booked
cols	A string containing the columns names for each row
empty_seats_per_row	A dictionary showing row numbers and their associated number of empty seats.

Outputs

Variable	**Description**
allocation_order	Sets a list of booking sizes that will be grouped together

Unit Tests

Test Description	UT1 tries to allocate 3 passengers to an empty airplane of width 4. A list containing the number 3 is expected to be returned
Results	Test Passed
Test Description	UT2 tries to allocate 5 people to an empty airplane of width 4. A list containing the numbers 4 and 1 are expected to be returned.
Results	Test Passed
Test Description	UT3 tries to allocate 4 seats to a partially booked airplane of width 4. A list containing the number 4 and 1 is expected to be returned.
Results	Test Passed

NOTE: The following functions were not unit tested as their functionality does not return values:

- organise_booking()
- write_database()
- clean_database()
- main()

Functional Testing

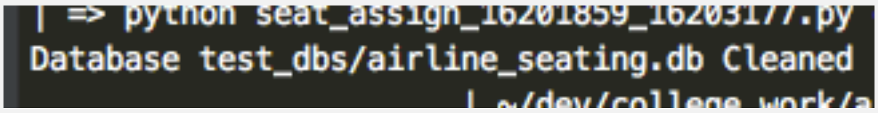
This section will test the overall functionality of the programme using a number of different database sizes and input booking files. The aim of these tests are to ensure that the overall programme is robust to different types of inputs and can deal correctly with poor inputs.

Clean database functionality

Test Case 1

Test that the clean database function works correctly.

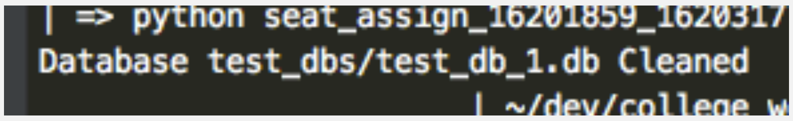
Input	airline_seating.db – A database constructed fully allocated with passengers.
Expected Outcomes	Standard message to be printed to the command line.

Results	Correct message printed to the command line:  `| => python seat_assign_16201859_16203177.py`
	`Database test_dbs/airline_seating.db Cleaned`
	`| ~/dev/college_work/a`
	seating table in airline_seating.db now has no passengers assigned and can be used by the program once again.

Test Case 2

Test the clean database function when asked to clean an already empty database

Input	test_db_1.db – A database constructed with no names currently assigned to any seats in the seating table
Expected Outcomes	Standard message to be printed to the command line. No errors to show.

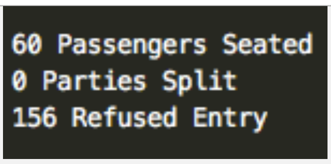
Results	Correct message printed to the command line:  `| => python seat_assign_16201859_1620317`
	`Database test_dbs/test_db_1.db Cleaned`
	`| ~/dev/college_w`

Passenger Assignment Functionality

The following test cases will test the overall program to ensure that it correctly allocates different size booking files with different size databases.

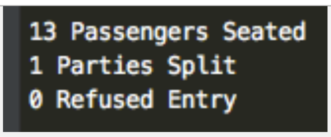
Test Case 1

Test Scenario	**Large Booking File/Small Seating Table**
Input	DB: empty airline_seating.db - 60 possible seats Booking file: bookings.csv – 217 potential bookings
Expected Outcomes	Passengers Refused: 156 Passengers Separated: 0

Results	
	`60 Passengers Seated`
	`0 Parties Split`
	`156 Refused Entry`
	
	Test Passed

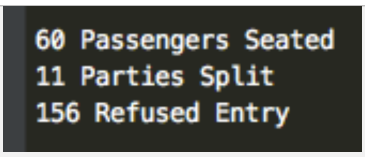
Test Case 2

Test Scenario	**Small Booking File/Small Seating Table**
Input	DB: empty airline_seating.db - 60 possible seats Booking file: test_csv/test_bookings.csv – 13 potential bookings
Expected Outcomes	Passengers Refused: 0 Passengers Separated: 1

Results	
	`13 Passengers Seated`
	`1 Parties Split`
	`0 Refused Entry`
	
	Test Passed

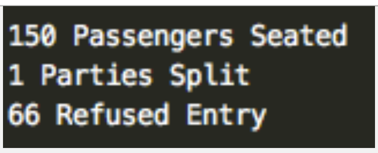
Test Case 3

Test Scenario	**Narrow seating structure**
Input	DB: test_db_2.db - 60 possible seats, rows are two seats wide (AB) Booking file: bookings.csv – 216 potential bookings
Expected Outcomes	Passengers Refused: 156 Passengers Separated: 11

Results	
	```
	60 Passengers Seated
	11 Parties Split
	156 Refused Entry
	```
	 Test Passed

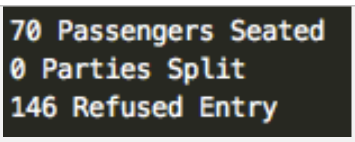
Test Case 4

Test Scenario	**Large Booking file / Large database**
Input	DB: test_db_3.db - 150 possible seats (row – ABCDE, 30 rows) Booking file: bookings.csv – 216 potential bookings
Expected Outcomes	Passengers Refused: 66 Passengers Separated: 1

Results	
	```
	150 Passengers Seated
	1 Parties Split
	66 Refused Entry
	```
	 Test Passed

Test Case 5

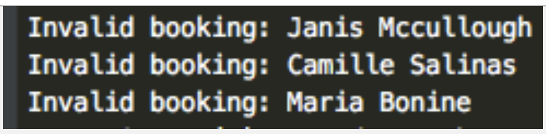
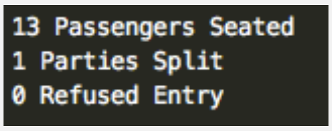
Test Scenario	**Wide and short seating structure**
Input	DB: test_db_4.db - 70 possible seats (row – ABCDEFG, 10 rows) Booking file: bookings.csv – 216 potential bookings
Expected Outcomes	Passengers Refused: 146 Passengers Separated: 0

Results	
	```
	70 Passengers Seated
	0 Parties Split
	146 Refused Entry
	```
	 Test Passed

Test Case 6

Test Scenario	**Erroneous input data**
Input	DB: airline_seating.db Booking file: test_bookings2.csv

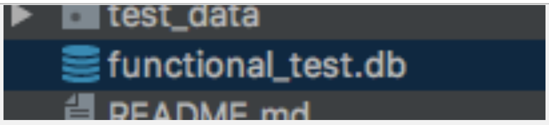
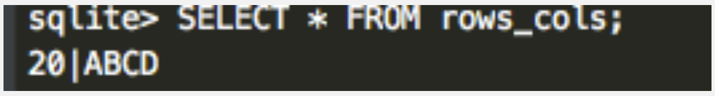
Expected Outcomes	Three “Invalid booking: <customer name>” messages Passengers Refused: 13 Passengers Separated: 1

Results	
	`Invalid booking: Janis Mccullough`
	`Invalid booking: Camille Salinas`
	`Invalid booking: Maria Bonine`
	
	`13 Passengers Seated`
	`1 Parties Split`
	`0 Refused Entry`
	
	Test Passed

Create database functionality

Test Case 1

Test Scenario	**Create test database**
Input	Db name: functional_test.db Rows: 20 Cols: “ABCD”
Expected Outcomes	Database name “functional_test.db” should be created with 20 rows and for columns names “A”, “B”, “C” and “D”

Results	
	test_data
	functional_test.db
	README.md
	
	`sqlite> SELECT * FROM rows_cols;`
	`20|ABCD`
	
	Test Passed