

MIS40750

Analytics Research
& Implementation



MSc Business Analytics

24th February 2017



Airline Seating Assignment

Group Members:

| | |
|---------------|----------|
| Eoin Carroll | 16202781 |
| Bobby Reardon | 13203828 |
| Chris Taylor | 13368376 |

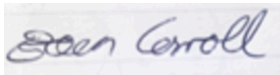

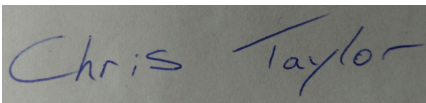
MIS40750

Airline Seating Assignment



STATEMENT OF AUTHORSHIP

"We declare that all of the undersigned have contributed to this work and that it is all our own work as understood by UCD policies on Academic Integrity and Plagiarism, unless otherwise cited".

| Name | | Number | Signature |
|---------------|--|----------|--|
| Eoin Carroll | | 16202781 |  |
| Bobby Reardon | | 13203828 |  |
| Chris Taylor | | 13368376 |  |

ABSTRACT

This project was undertaken to create a python program that assigns seating to a list of passengers for an airline. In a team of three, we have created a robust software application that takes inputs as required and produces an update SQL database as an output. This report documents the assumptions, testing details and user documentation for the implementation.

TABLE OF CONTENTS

| | |
|------------------------------------|---|
| 1. INTRODUCTION | 1 |
| 2. CODING ASSUMPTIONS | 1 |
| 2.1 Rectangular Plane | |
| 2.2 Ignoring Aisles | |
| 2.3 Existing Bookings | |
| 2.4 Passengers Separated Metric | |
| 2.5 Priority of Bookings | |
| 2.6 Database Ordering | |
| 2.7 First Come First Served | |
| 2.8 Bookings with Same Name | |
| 3. CODING STRUCTURE | 3 |
| 3.1 Run Time & Machine Resources | |
| 3.2 Robustness | |
| 3.3 Simplicity | |
| 4. DESCRIPTION OF USER COMMANDS | 4 |
| 5. TROUBLESHOOTING | 4 |
| 6. TESTING | 5 |
| def test_count_list(self) | |
| def test_count_str_list(self) | |
| def test_read_database(self) | |
| def test_read_csv(self) | |
| def test_assign_metrics_list(self) | |

| | |
|---------------------------|---|
| 7. REMAINING ISSUES | 7 |
| 8. CONCLUSION | 7 |
| 9. FUNCTION DOCUMENTATION | 8 |
| count_str_list | |
| count_list | |
| assign_metrics_list | |
| run_all | |
| read_database | |
| read_csv | |

LIST OF FIGURES

- 1: Plane Layout
- 2: Inputted Database
- 3: Updated Database

1. INTRODUCTION

This assignment simulates a real world problem of an airline seating system. It was required of us to write a python program to read a plane's seating layout from an SQL database, take a CSV list of booking information and then implement an algorithm to seat as many passengers as possible while separating as few booking groups as possible. The assignment had a number of required outputs saved to an SQL database - the booking names in each seat, number of passengers rejected and number of groups separated.

A program was successfully created that meets the assignment brief in a team of three. This document is a record of the thought process in creating the program, the approach taken, the obstacles overcome, the testing performed and a conclusion that relates back to the learning outcomes. The appendix includes detail user documentation for each function.

2. CODING ASSUMPTIONS

Due to the variety of factors and scenarios to be accounted for when developing the program, several assumptions were made to overcome them or to simplify the functionality. What follows are a list of the main assumptions made by the program and any changes made to remove them, beginning with the shape of the plane.

2.1 Rectangular plane: Based on the row_cols table of the sample database and to increase the simplicity of the model, it was assumed that the seat layout would be rectangular and wouldn't include the possibility of rows with fewer/more seats than others. Figure 1 overleaf shows the plane layout generated from the sample database.

2.2 Ignoring Aisles: The sample database gave no information about the location of aisle(s) on the plane. This could be an issue for planes with a high number of seats per row as there would be several possible configurations. Hence, in order to generalise for all planes, the aisles were taken to be irrelevant with regard to splitting passengers. As a result, a couple assigned to the seats 3C and 3D, as per the red group in figure 1, would be considered in the model to be beside each other and not separated.

2.3 Existing Bookings: Initially our program was designed such that it assumed no bookings existed prior to running the algorithm. However, this has been corrected and the program now adapts to this situation. In the sample database, seats 1A and 1C were already assigned to Donald Trump and Hillary Clinton respectively, as seen in the blue seats in figure 1. As a result, our algorithm assigns the remaining 58 seats to passengers in the bookings CSV file.

However, a further assumption still being made is that the existing bookings are not spaced out and if they are in different rows, they at least operate from left to right (A → F) in each row

2.4 Passengers Separated Metric: The definition provided for this metric is “a number representing how many passengers are seated away from any other member of their party”. Following discussion of how this should be interpreted, we assumed by default that this metric should represent the total number of passengers in a booking, if that booking has been split at all. In this case, the red group would lead to an increase of 0, while the yellow and green groups would increase the metric by 6 and 7 respectively. However, the model has been adapted to take in user specified interpretations. The alternatives for this metric are the *Alone*, *Separated* and *Dissatisfaction* metrics. The *Alone* interpretation increases the metric by the number of passengers in a booking who get seated completely alone from the rest of their party. In this case, the green group would lead to an increase of 0, while the yellow group would increase the metric by 1. Next is the *Separated* metric which counts the total number of group separations. Under this interpretation, the metric would be increased by 1 for the green booking in figure 1 and increased by 2 for the yellow group. The *Dissatisfaction* metric is an alternative measure which applies a football metric approach. The metric is increased by 1 in a situation where a group has been separated but no passenger is alone (such as the green group). This is irrelevant of how many splits there are. If one or more passengers in a group are sitting entirely alone (such as in the yellow group) then the metric is increased by 3, thus penalising separations of this kind more heavily. These interpretations can be called by the user by setting the *sep* parameter to *Total* (default), *Alone*, *Separated* or *Dissatisfaction* as required.

2.5 Priority of Bookings: It was intuitively assumed that the bookings should be processed in order of the rows in the provided CSV file. It can be modified however as the program can take inputs for the first and last rows to be processed in a single run of the program. This allows for the user to run the program for a block of bookings in isolation, e.g. bookings 10 through 20. Further information about these parameters are included in the function documentation.

2.6 Database Ordering: Initially the algorithm was designed in such a way that it assumed the ordering of the rows in the seating

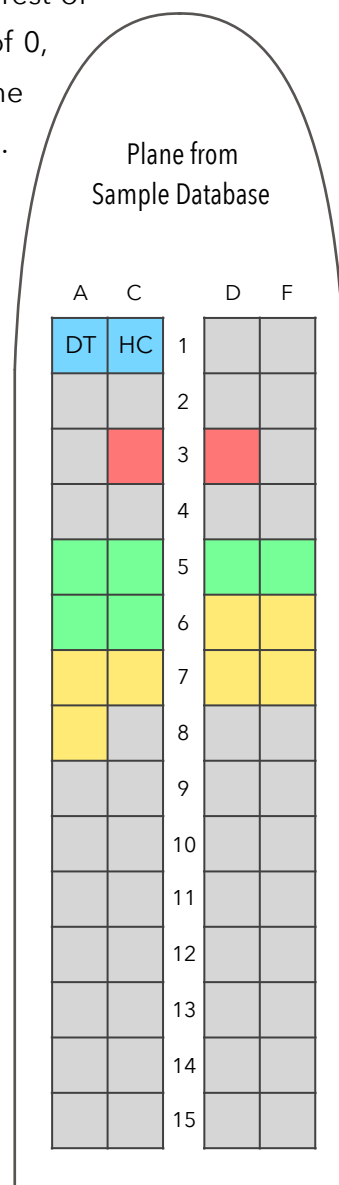


Figure 1: Plane Layout

table of the database would be the same for any other databases tested. However, this assumption has been corrected as it now identifies seats by row and seat number and hence, it is independent of the original order of database tables.

2.7 First Come First Served: Unlike a lot of booking systems which intentionally disallow bookings that would result in a single seat being left alone, our program proceeds anyway. As a result, a booking of two people will be accommodated in a row with three free seats, ignoring the fact that this isolates a seat. This simplifies the seat assigning process, but doesn't create many problems due to the large number of bookings of size 1 which will fill those places. It does however work in a greedy front to back manner, and will place the two person booking in the three seats, even when rows further back might have even numbers of seats available.

2.8 Bookings with Same Name: Initially the algorithm wasn't able to deal with multiple bookings of the same name and assumed this would not occur. However, it is no longer assumed to be the case. Should a booking be assigned with the name "Donald Trump", the same as the existing booking for seat 1A in figure 1, the booking name is modified to "Donald Trump (1)" and processed. Further bookings of the same name will be changed to "Donald Trump (1)(1)" and so on.

3. CODING STRUCTURE

The overall structure of our program is designed to minimise run time and machine resources while being as robust but simple as possible.

3.1 Run Time and Machine Resources: An example of minimising run time is storing the number of free seats in memory and only attempting to seat passengers if we know there is space available. In addition to this, we revised functions within the program a number of times to simplify and minimise computational resources. One example being originally utilising numpy arrays requiring importing a library - we have now switched to a simplified list to keep track of passenger names.

3.2 Robustness: The first example of the robustness of our program is storing a default value for most function input arguments. By storing the most commonly used database and CSV filenames, we can make an assumption that the user wants to run the program with standard input but also allows unique file names to be accepted. Another example of this is our interpretation of the separation metric. A second robustness example is flexibility to handle any seating arrangement in a plane. All functions use variables from the plane layout database input to process the information appropriately. A third example is handling the occurrence of two bookings having the same name. The program performs a check and can assign an incremental

integer to each group booking with the same name. For example we can now seat three separate groups called "Donald Trump", as "Donald Trump", "Donald Trump (1)" and "Donald Trump (1)(1)". This prevents confusion and errors in seating arrangements.

3.3 Simplicity: The program is logically laid out with a number of easily understandable functions. We have annotated the functions to show what they do and how they work which allowed more team collaboration during development and now allows external users analyse the code. We have used the most basic python structures where possible such as lists in order to simplify error checking and stability across different systems and environments.

4. DESCRIPTION OF USER COMMANDS

The `run_all` function is run by the program when called from the command line and uses default values. However, the user of the program can make several specifications. The program by default runs for a database file called `'airline_seating.db'` and a booking list file called `'bookings.csv'`. These can be replaced with files of the same format if desired. Three interpretations of the `passengers_separated` metric are included in the program. The default is `'Total'` but other interpretations can be specified by modifying the `sep` parameter. Further details are provided in the description of the `assign_metrics_list` function in the appendices. The assigning function operates one booking at a time, and so the user is provided the control to select the beginning and ending rows of the bookings file to process. By default, the entire CSV file is processed but the user can modify this if they desire.

5. TROUBLESHOOTING

During the coding process, there were several issues with cross platform IDEs. Bobby was using Enthought Canopy, 1.7.4.3348 (64 bit) on macOS Sierra 10.12.1, where the program ran successfully. However, Eoin running Windows 7 and 10 64bit, Spyder 2.3.9, iPython 4.2.0 and Python 3.5.2 from Anaconda 4.1.1. Meanwhile, Chris was running PyCharm 2016.2.3 on Windows 8 and similar errors occurred. Among the issues that arose were:

- np array showed (B,data) for strings inside numpy array
- IDE needed explicit instruction for `np.zeros` and `np.shape`

As a result of the issues involving the use of arrays, it was decided to alter the code to operate using lists instead.

6. TESTING

In order to test the program, test class was split into four sections, testing each of the functions that have return values. As the functions act differently based on the value of the input arguments, the following variables were defined at the top of the test class, purely for the purpose of testing:

```
sample_plane_list = ['', '', '', '']

sample_plane_list2 = ['Donald Trump', 'Donald Trump', '', '']

sample_plane_list3 = ['Donald Trump', 'Hilary Clinton',
                      'Hilary Clinton', 'Hilary Clinton', '', 'Hilary Clinton', '', '']

sample_row_length = 4
sample_string = 'Donald Trump'

sample_string2 = 'Hilary Clinton'

sample_integer = 0
sample_integer2 = 1
```

```
def test_count_list(self):
```

As stated in its description, this function is designed to count the number of available seats in a given row. Three aspects were identified for testing this:

- Base case: Given `sample_plane_list`, the function did indeed return the desired output of four available seats.
- Given `sample_plane_list2` the function identified two seats as containing passengers, and correctly returned that the remaining two were free.
- To ensure the function worked in cases with more than one row, two `assertEqual` tests were run on `sample_plane_list3`. Initially, the first row was tested (which appears as every second entry to the list), and the function accurately found two free spaces. Subsequently, when the second row was tested, the function found a single free space as expected.

```
def test_count_str_list(self):
```

As stated in its description, this function is designed to count the occurrence of a specific string in the plane. Bearing this in mind, three assertions were used to check that the function could handle different cases:

- Using `sample_plane_list`, it was checked that employing the function on an empty plane would indeed return the value of zero for a specific string's occurrence.

- Using `sample_plane_list2`, it was checked that the function returned the correct occurrence value when there were occurrences of the string in the plane.
- Using `sample_plane_list3`, it was ensured that the the function still returned the correct value given that a group had been split up and thus no longer seated in sequence.
- Upon completion, each of the above tests returned the expected value, and hence it was considered the function to be robust.

```
def test_read_database(self):
```

In order to test that the database would be read correctly, it was suitable to test the function on a sample database file with similar tables but different values. With this in mind, a database was created with similar tables, using ten rows with three seats per row (X,Y,Z) instead of the fifteen rows with four seats (A,C,D,F). Having run the function on this test-database, the expected integers 10 (rows) / 30 (seats), and the expected 'XYZ' string were returned.

```
def test_read_csv(self):
```

Similar to the prior test - it was decided that a sample file was required to perform an accurate test on the function for reading .csv files. Contained within the submission folder, there is a file `test_booking.csv` containing three names (Chris, Bobby, Eoin) and a number of booked seats for each. Thus, in the test class, it is asserted that calling the function on this file would return: (3, ['Bobby', 'Chris', 'Eoin'], ['2', '4', '3']). Having passed this test, the function is considered to be robust.

```
def test_assign_metrics_list(self):
```

As this function hasn't got a return value, a more practical test was used for testing. The purpose of the function is to update the database, so the sample database (as seen in figure 2) and a sample .csv file were inputted into the program to check the effect it would have on the database. The function updated the database as designed, as can be seen in figure 3.

Figure 3: Inputted Database

| | row | seat | name |
|----|--------|--------|--------|
| | Filter | Filter | Filter |
| 1 | 1 | X | |
| 2 | 2 | X | |
| 3 | 3 | X | |
| 4 | 4 | X | |
| 5 | 5 | X | |
| 6 | 6 | X | |
| 7 | 7 | X | |
| 8 | 8 | X | |
| 9 | 9 | X | |
| 10 | 10 | X | |
| 11 | 1 | Y | |
| 12 | 2 | Y | |
| 13 | 3 | Y | |
| 14 | 4 | Y | |
| 15 | 5 | Y | |

Figure 3: Updated Database

| | row | seat | name |
|----|--------|--------|--------|
| | Filter | Filter | Filter |
| 1 | 1 | X | Bobby |
| 2 | 2 | X | Chris |
| 3 | 3 | X | Eoin |
| 4 | 4 | X | |
| 5 | 5 | X | |
| 6 | 6 | X | |
| 7 | 7 | X | |
| 8 | 8 | X | |
| 9 | 9 | X | |
| 10 | 10 | X | |
| 11 | 1 | Y | Bobby |
| 12 | 2 | Y | Chris |
| 13 | 3 | Y | Eoin |
| 14 | 4 | Y | |
| 15 | 5 | Y | |

7. REMAINING ISSUES

While the program now operates as detailed in the assignment specification, there is one primary issue that remains that has not been overcome. This issue concerns the spacing of existing bookings in the inputted database file. The program still assumes that all prior bookings are located in a left to right manner within each row, without any spaces between them. Should this assumption be broken, there is a high likelihood that existing bookings (or parts thereof) will get overwritten due to the structure of the assigning algorithm.

8. CONCLUSION

The submitted program runs correctly and has been rigorously tested using a variety of input simulations. The final program version is the result of iterative collaboration and teamwork during the project. The Github repository is a record of the contribution that each member made and the number of modifications that were performed in order to optimise the software. We have been conscious of the need to record assumptions, user documentation and annotate the program to simulate a real world software project. The project brief has been successfully interpreted, a solution has been implemented to meet the defined criteria and the project has been recorded in this document.

9. FUNCTION DOCUMENTATION

count_str_list

Description

`count_str_list` is used to count the number of entries of a list equal to a given string.

Usage

```
count_str_list(A, str)
```

Arguments

A

A list representing the names assigned to each seat on the plane. This list is assumed to be in the same format as the original seating table of the database, with entries ordered by seat letter, then by row number (ie, 1A, 2A, ... 15A, 1B, ...).

str

String representing the name that is being checked.

Details

`count_str_list` is used within `assign_metrics_list` to count the number of seats assigned to a passenger with the booking name `str`. It again applies the same rectangular plane assumption as the rest of the program.

Outputs

`count_str_list` returns the following outputs:

c

Integer representing the number of seats on the plane with booking name `str`.

Author(s)

Bobby Reardon

See Also

`count_str_list`, which is also used within `assign_metrics_list`.

Examples

```
A_list = ['']*60
```

```
A_list[8] = 'Joe Bloggs'
```

```
In[1]: count_str_list(A_list, 'Joe Bloggs')
```

```
Out[1]: 1
```

count_list

Description

`count_list` is used to count the number of empty (ie. ' ') seats in a given row of a given plane.

Usage

```
count_list(A, rowlen, i)
```

Arguments

A

A list representing the names assigned to each seat on the plane. This list is assumed to be in the same format as the seating table of the database, with entries ordered by seat letter, then by row number (ie, 1A, 2A, ... 15A, 1B, ...).

rowlen

Integer representing the number of seats in each row of the plane.

i

An integer representing the row of interest, where 0 represents the first row.

Details

`count_list` is used within `assign_metrics_list` to count the number of available seats in a given row by counting the number of empty ' ' entries. It applies the same rectangular plane assumption as the rest of the program.

Outputs

`count_list` returns the following outputs:

c

Integer representing the number of free/available seats remaining in row i.

Author(s)

Bobby Reardon

See Also

`count_str_list`, which is also used within `assign_metrics_list`.

Examples

```
A_list = [' ']*60
```

```
In[1]: count_list(A_list, 4, 2)
```

```
Out[1]: 4
```

assign_metrics_list

Description

`assign_metrics_list` is used to allocate seats, that are currently unoccupied on the plane, to a booking of given size and name.

Usage

```
assign_metrics_list(db, booking_name, booking_size, sep = 'Total')
```

Arguments

`db`

A string representing the database file name where the details of the plane are stored, along with the current values of the metrics.

`booking_name`

A string representing the name used for the booking which will be use to designate the seats once the assigning is completed.

`booking_size`

An integer representing the size of the party to which seats will be assigned.s

`sep`

A string representing the interpretation of the passengers_separated metric to be used. This can take the inputs: `'Separations'`, `'Alone'`, `'Total'` [default] and `'Dissatisfaction'`. See 'Details' for more.

Details

`assign_metrics_list` first accesses the database file `db`. It retrieves the dimensions of the plane from the `rows_cols` table and creates a list of names for each seat from the `seating` table.

A while loop is run to check if any prior bookings have the same name as the one being processed. Until a unique name is generated, the loop adds `' (1) '` to the end of the booking name so as to run the assign step correctly.

The assigning algorithm operates by trying to fit the largest possible group together in a single row, each time scanning rows from front to back. In the event that no row can accommodate the booking size, the booking is partitioned by separating a single individual and then attempting to assign seats to the slightly smaller booking size. The functions `count_list` and `count_str_list` (see below) are used in this process. The process iterates until it allocates seats to the largest possible subgroup of the original booking. Once completed it repeats the process if necessary for the remaining members of the party. At each step of the process, the seating table of the database file is updated.

Assigning step assumes all seats already booked have been done so in a left-to-right manner with no gaps between occupied seats within a given row.

A list of the row numbers of each occupied seat is then generated and this is then used in different manners to update the `passengers_separated` metric depending on the choice of interpretation.

Interpretations:

- `'Alone'` interprets the metric as the number of passengers seated who are completely separated from all other members of the party.

- `'Separated'` interprets the metric as the total number of party splits made when assigning the seats.

- `'Total'` [default] interprets the metric as the total people sitting away from any other member of their party, ie. if a group is split, it is equal to the group size.

- `'Dissatisfaction'` interprets the metric by assigning dissatisfaction weightings to each booking which are defined as:

 - 0 if all party members are seated together.

 - 1 if party members are separated but no individuals are sat alone

 - 3 if one or more party members are sat alone.

Outputs

`assign_metrics_list` returns a printed statement `'Booking Confirmed'` once the database file has been updated correctly

Author(s)

Bobby Reardon

See Also

`count_list` and `count_str_list` functions used in `assign_metrics_list`.

Examples

```
import sqlite3
import csv
```

```
DB_file = "airline_seating.db"
booking_name = "Joe Bloggs"
booking_size = 6
```

```
assign_metrics_list(DB_file, booking_name, booking_size, 'Alone')
```

```
# Using default value of sep:
assign_metrics_list('plane.db', 'John Murphy', 4)
```

run_all

Description

run_all is used to run the entire program while allowing the user to specify an interpretation of the separations metric as well as providing the user control over which entries in the bookings file are run.

Usage

```
run_all(DB_file = 'airline_seating.db', CSV_file = 'bookings.csv', sep = 'Total', first=1, last=0)
```

Arguments

DB_file

A string representing the database file name where the details of the plane are stored, along with the current values of the metrics. By default it will load the database file 'airline_seating.db' as per the sample file.

CSV_file

A string representing the bookings CSV file name where the ordered list of bookings are stored, with both booking names and corresponding sizes recorded. By default it will load the CSV file 'bookings.csv' as per the sample file.

sep

A string representing the interpretation of the passengers_separated metric to be used. This can take the inputs: 'Separations', 'Alone', 'Total' [default] and 'Dissatisfaction'. See 'Details' for more.

first

An integer representing the row number (1 indexed) of the bookings file for the algorithm to start at. By default, the algorithm begins with the first row.

last

An integer representing the last number (1 indexed) of the bookings file that the algorithm will run. By default, the algorithm ends with the last row.

Details

run_all begins by calling both the read_database and read_csv functions in order to extract the relevant information. It then sets the first and last rows of the CSV file to be processed.

For each of the k (end - start) bookings to be processed, the function begins by evaluating whether the plane can still accommodate the entire booking size. If it cannot be processed, the passengers_refused metric is updated and the new value is printed to screen. If it can be

processed, the `assign_metrics_list` function is called and the seating table and `passengers_separated` metric within the database are subsequently updated.

Outputs

`run_all` returns a printed statement for each booking which was accommodated and assigned along with the size of the booking. If the booking has failed to be assigned, the function prints a message to confirm no more free seats exist and prints the updated `passengers_refused` metric.

Author(s)

Bobby Reardon, Eoin Carroll

See Also

`run_all` comprises of three constituent functions, namely `readDB`, `readCSV` and `assign_metrics_list`. See these for more details.

Examples

```
import sqlite3
import csv
```

```
DB_file = "airline_seating.db"
CSV_file = "bookings.csv"
```

```
# Run entire bookings list with default metric interpretation:
run_all(DB_file, CSV_file)
```

```
# Or alternatively
run_all()
```

```
# Run first 10 entires using 'Alone' interpretation:
run_all(DB_file, CSV_file, 'Alone', 1, 10)
```

read_database

Description

read_database is a function that reads the seat layout from a database file to record the layout of the plane. It also checks to see how many free seats are available.

Usage

```
read_database(db_file = "airline_seating.db")
```

Arguments

db_file

The input argument represents the filename of the database to be used. It defaults to "airline_seating.db" for the purpose of this assignment.

Details

This function utilises the `sqlite3` library to interact with an SQL database. It opens the specified file and returns specific values from different tables in the database. It also iterates through the seat table and calculates the number of free seats.

We assume that the database is set up correctly as per the sample database provided for this assignment.

Outputs

read_database returns the following outputs:

number_of_rows

Represents the number of rows on the plane taken from the database.

seat_layout

Represents the seating format in the form of seat letters per row.

free_seats

Represents the number of unoccupied seats in the database.

Author

Eoin Carroll

Example

```
import sqlite3
```

```
DB_file = "airline_seating.db"  
nrows, seat_layout, free_seats = read_database(DB_file)
```

read_csv

Description

`read_csv` is a function that reads the list of passengers that wish to be booked onto the plane from a CSV file.

Usage

```
read_csv(csv_file = "bookings.csv")
```

Arguments

`csv_file`

The input argument represents the filename of the CSV file to be used. It defaults to `"bookings.csv"` for the purpose of this assignment.

Details

This function utilises the CSV library to interact with the CSV file. It opens the specified file and reads the name and number for each passenger booking.

We assume that the CSV file is set up correctly as per the sample file provided for this assignment. The assumed format contains the passenger names in column 1 and the passenger group sizes in column 2.

Outputs

`read_csv` returns the following outputs:

`booking_number`

Represents the number of unique booking requests

`booking_name`

Represents the name of each booking. List format.

`booking_size`

Represents the size of each booking. List format.

Author

Eoin Carroll

Example

```
import csv
```

```
CSV_file = "bookings.csv"
```

```
booking_number, booking_name, booking_size = read_csv(CSV_file)
```

1. INTRODUCTION

This assignment simulates a real world problem of an airline seating system. It was required of us to write a python program to read a plane's seating layout from an SQL database, take a CSV list of booking information and then implement an algorithm to seat as many passengers as possible while separating as few booking groups as possible. The assignment had a number of required outputs saved to an SQL database - the booking names in each seat, number of passengers rejected and number of groups separated.

A program was successfully created that meets the assignment brief in a team of three. This document is a record of the thought process in creating the program, the approach taken, the obstacles overcome, the testing performed and a conclusion that relates back to the learning outcomes. The appendix includes detail user documentation for each function.

2. CODING ASSUMPTIONS

Due to the variety of factors and scenarios to be accounted for when developing the program, several assumptions were made to overcome them or to simplify the functionality. What follows are a list of the main assumptions made by the program and any changes made to remove them, beginning with the shape of the plane.

2.1 Rectangular plane: Based on the row_cols table of the sample database and to increase the simplicity of the model, it was assumed that the seat layout would be rectangular and wouldn't include the possibility of rows with fewer/more seats than others. Figure 1 overleaf shows the plane layout generated from the sample database.

2.2 Ignoring Aisles: The sample database gave no information about the location of aisle(s) on the plane. This could be an issue for planes with a high number of seats per row as there would be several possible configurations. Hence, in order to generalise for all planes, the aisles were taken to be irrelevant with regard to splitting passengers. As a result, a couple assigned to the seats 3C and 3D, as per the red group in figure 1, would be considered in the model to be beside each other and not separated.

2.3 Existing Bookings: Initially our program was designed such that it assumed no bookings existed prior to running the algorithm. However, this has been corrected and the program now adapts to this situation. In the sample database, seats 1A and 1C were already assigned to Donald Trump and Hillary Clinton respectively, as seen in the blue seats in figure 1. As a result, our algorithm assigns the remaining 58 seats to passengers in the bookings CSV file.

However, a further assumption still being made is that the existing bookings are not spaced out and if they are in different rows, they at least operate from left to right (A → F) in each row

2.4 Passengers Separated Metric: The definition provided for this metric is “a number representing how many passengers are seated away from any other member of their party”. Following discussion of how this should be interpreted, we assumed by default that this metric should represent the number of group separations. Under this interpretation, the metric would be increased by 1 for the green booking in figure 1 and increased by 2 for the yellow group. However, the model has been adapted to take in user specified interpretations. The alternatives for this metric are the *Alone*, *Total* and *Dissatisfaction* metrics. The *Alone* interpretation increases the metric by the number of passengers in a booking who get seated completely alone from the rest of their party. In this case, the green group would lead to an increase of 0, while the yellow group would increase the metric by 1. The *Total* interpretation increases the metric by the total number of passengers in a booking, if that booking has been split at all. In this case, the red group would lead to an increase of 0, while the yellow and green groups would increase the metric by 6 and 7 respectively. The *Dissatisfaction* metric is an alternative measure which applies a football metric approach. The metric is increased by 1 in a situation where a group has been separated but no passenger is alone (such as the green group). This is irrelevant of how many splits there are. If one or more passengers in a group are sitting entirely alone (such as in the yellow group) then the metric is increased by 3, thus penalising separations of this kind more heavily. These interpretations can be called by the user by setting the *sep* parameter to ‘Separations’ (default), ‘Alone’ or ‘Dissatisfaction’ as required.

2.5 Priority of Bookings: It was intuitively assumed that the bookings should be processed in order of the rows in the provided CSV file. It can be modified however as the program can take inputs for the first and last rows to be processed in a single run of the program. This allows for the user to run the program for a block of bookings in isolation, e.g. bookings 10 through 20. Further information about these parameters are included in the function documentation.

2.6 Database Ordering: Initially the algorithm was designed in such a way that it assumed the ordering of the rows in the seating

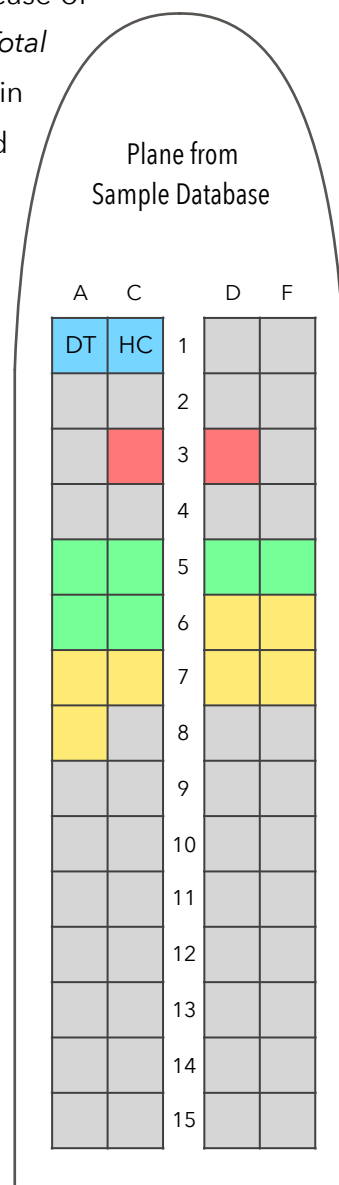


Figure 1: Plane Layout

table of the database would be the same for any other databases tested. However, this assumption has been corrected as it now identifies seats by row and seat number and hence, it is independent of the original order of database tables.

2.7 First Come First Served: Unlike a lot of booking systems which intentionally disallow bookings that would result in a single seat being left alone, our program proceeds anyway. As a result, a booking of two people will be accommodated in a row with three free seats, ignoring the fact that this isolates a seat. This simplifies the seat assigning process, but doesn't create many problems due to the large number of bookings of size 1 which will fill those places. It does however work in a greedy front to back manner, and will place the two person booking in the three seats, even when rows further back might have even numbers of seats available.

2.8 Bookings with Same Name: Initially the algorithm wasn't able to deal with multiple bookings of the same name and assumed this would not occur. However, it is no longer assumed to be the case. Should a booking be assigned with the name "Donald Trump", the same as the existing booking for seat 1A in figure 1, the booking name is modified to "Donald Trump (1)" and processed. Further bookings of the same name will be changed to "Donald Trump (1)(1)" and so on.

3. CODING STRUCTURE

The overall structure of our program is designed to minimise run time and machine resources while being as robust but simple as possible.

3.1 Run Time and Machine Resources: An example of minimising run time is storing the number of free seats in memory and only attempting to seat passengers if we know there is space available. In addition to this, we revised functions within the program a number of times to simplify and minimise computational resources. One example being originally utilising numpy arrays requiring importing a library - we have now switched to a simplified list to keep track of passenger names.

3.2 Robustness: The first example of the robustness of our program is storing a default value for most function input arguments. By storing the most commonly used database and CSV filenames, we can make an assumption that the user wants to run the program with standard input but also allows unique file names to be accepted. Another example of this is our interpretation of the separation metric. A second robustness example is flexibility to handle any seating arrangement in a plane. All functions use variables from the plane layout database input to process the information appropriately. A third example is handling the occurrence of two bookings having the same name. The program performs a check and can assign an incremental

integer to each group booking with the same name. For example we can now seat three separate groups called "Donald Trump", as "Donald Trump", "Donald Trump (1)" and "Donald Trump (1)(1)". This prevents confusion and errors in seating arrangements.

3.3 Simplicity: The program is logically laid out with a number of easily understandable functions. We have annotated the functions to show what they do and how they work which allowed more team collaboration during development and now allows external users analyse the code. We have used the most basic python structures where possible such as lists in order to simplify error checking and stability across different systems and environments.

4. DESCRIPTION OF USER COMMANDS

The `run_all` function is run by the program when called from the command line and uses default values. However, the user of the program can make several specifications. The program by default runs for a database file called `'airline_seating.db'` and a booking list file called `'bookings.csv'`. These can be replaced with files of the same format if desired. Three interpretations of the `passengers_separated` metric are included in the program. The default is `'Separated'` but other interpretations can be specified by modifying the `sep` parameter. Further details are provided in the description of the `assign_metrics_list` function in the appendices. The assigning function operates one booking at a time, and so the user is provided the control to select the beginning and ending rows of the bookings file to process. By default, the entire CSV file is processed but the user can modify this if they desire.

5. TROUBLESHOOTING

During the coding process, there were several issues with cross platform IDEs. Bobby was using Enthought Canopy, 1.7.4.3348 (64 bit) on macOS Sierra 10.12.1, where the program ran successfully. However, Eoin running Windows 7 and 10 64bit, Spyder 2.3.9, iPython 4.2.0 and Python 3.5.2 from Anaconda 4.1.1. Meanwhile, Chris was running PyCharm 2016.2.3 on Windows 8 and similar errors occurred. Among the issues that arose were:

- np array showed (B,data) for strings inside numpy array
- IDE needed explicit instruction for `np.zeros` and `np.shape`

As a result of the issues involving the use of arrays, it was decided to alter the code to operate using lists instead.

6. TESTING

In order to test the program, test class was split into four sections, testing each of the functions that have return values. As the functions act differently based on the value of the input arguments, the following variables were defined at the top of the test class, purely for the purpose of testing:

```
sample_plane_list = ['', '', '', '']

sample_plane_list2 = ['Donald Trump', 'Donald Trump', '', '']

sample_plane_list3 = ['Donald Trump', 'Hilary Clinton',
                      'Hilary Clinton', 'Hilary Clinton', '', 'Hilary Clinton', '', '']

sample_row_length = 4
sample_string = 'Donald Trump'

sample_string2 = 'Hillary Clinton'

sample_integer = 0
sample_integer2 = 1
```

def test_count_list(self):

As stated in its description, this function is designed to count the number of available seats in a given row. Three aspects were identified for testing this:

- Base case: Given `sample_plane_list`, the function did indeed return the desired output of four available seats.
- Given `sample_plane_list2` the function identified two seats as containing passengers, and correctly returned that the remaining two were free.
- To ensure the function worked in cases with more than one row, two `assertEqual` tests were run on `sample_plane_list3`. Initially, the first row was tested (which appears as every second entry to the list), and the function accurately found two free spaces. Subsequently, when the second row was tested, the function found a single free space as expected.

def test_count_str_list(self):

As stated in its description, this function is designed to count the occurrence of a specific string in the plane. Bearing this in mind, three assertions were used to check that the function could handle different cases:

- Using `sample_plane_list`, it was checked that employing the function on an empty plane would indeed return the value of zero for a specific string's occurrence.

- Using `sample_plane_list2`, it was checked that the function returned the correct occurrence value when there were occurrences of the string in the plane.
- Using `sample_plane_list3`, it was ensured that the the function still returned the correct value given that a group had been split up and thus no longer seated in sequence.
- Upon completion, each of the above tests returned the expected value, and hence it was considered the function to be robust.

```
def test_read_database(self):
```

In order to test that the database would be read correctly, it was suitable to test the function on a sample database file with similar tables but different values. With this in mind, a database was created with similar tables, using ten rows with three seats per row (X,Y,Z) instead of the fifteen rows with four seats (A,C,D,F). Having run the function on this test-database, the expected integers 10 (rows) / 30 (seats), and the expected 'XYZ' string were returned.

```
def test_read_csv(self):
```

Similar to the prior test - it was decided that a sample file was required to perform an accurate test on the function for reading .csv files. Contained within the submission folder, there is a file `test_booking.csv` containing three names (Chris, Bobby, Eoin) and a number of booked seats for each. Thus, in the test class, it is asserted that calling the function on this file would return: (3, ['Bobby', 'Chris', 'Eoin'], ['2', '4', '3']). Having passed this test, the function is considered to be robust.

```
def test_assign_metrics_list(self):
```

As this function hasn't got a return value, a more practical test was used for testing. The purpose of the function is to update the database, so the sample database (as seen in figure 2) and a sample .csv file were inputted into the program to check the effect it would have on the database. The function updated the database as designed, as can be seen in figure 3.

Figure 3: Inputted Database

| | row | seat | name |
|----|--------|--------|--------|
| | Filter | Filter | Filter |
| 1 | 1 | X | |
| 2 | 2 | X | |
| 3 | 3 | X | |
| 4 | 4 | X | |
| 5 | 5 | X | |
| 6 | 6 | X | |
| 7 | 7 | X | |
| 8 | 8 | X | |
| 9 | 9 | X | |
| 10 | 10 | X | |
| 11 | 1 | Y | |
| 12 | 2 | Y | |
| 13 | 3 | Y | |
| 14 | 4 | Y | |
| 15 | 5 | Y | |

Figure 3: Updated Database

| | row | seat | name |
|----|--------|--------|--------|
| | Filter | Filter | Filter |
| 1 | 1 | X | Bobby |
| 2 | 2 | X | Chris |
| 3 | 3 | X | Eoin |
| 4 | 4 | X | |
| 5 | 5 | X | |
| 6 | 6 | X | |
| 7 | 7 | X | |
| 8 | 8 | X | |
| 9 | 9 | X | |
| 10 | 10 | X | |
| 11 | 1 | Y | Bobby |
| 12 | 2 | Y | Chris |
| 13 | 3 | Y | Eoin |
| 14 | 4 | Y | |
| 15 | 5 | Y | |

7. REMAINING ISSUES

While the program now operates as detailed in the assignment specification, there is one primary issue that remains that has not been overcome. This issue concerns the spacing of existing bookings in the inputted database file. The program still assumes that all prior bookings are located in a left to right manner within each row, without any spaces between them. Should this assumption be broken, there is a high likelihood that existing bookings (or parts thereof) will get overwritten due to the structure of the assigning algorithm.

8. CONCLUSION

The submitted program runs correctly and has been rigorously tested using a variety of input simulations. The final program version is the result of iterative collaboration and teamwork during the project. The Github repository is a record of the contribution that each member made and the number of modifications that were performed in order to optimise the software. We have been conscious of the need to record assumptions, user documentation and annotate the program to simulate a real world software project. The project brief has been successfully interpreted, a solution has been implemented to meet the defined criteria and the project has been recorded in this document.

9. FUNCTION DOCUMENTATION

`count_str_list`

Description

`count_str_list` is used to count the number of entries of a list equal to a given string.

Usage

```
count_str_list(A, str)
```

Arguments

A

A list representing the names assigned to each seat on the plane. This list is assumed to be in the same format as the original seating table of the database, with entries ordered by seat letter, then by row number (ie, 1A, 2A, ... 15A, 1B, ...).

str

String representing the name that is being checked.

Details

`count_str_list` is used within `assign_metrics_list` to count the number of seats assigned to a passenger with the booking name `str`. It again applies the same rectangular plane assumption as the rest of the program.

Outputs

`count_str_list` returns the following outputs:

c

Integer representing the number of seats on the plane with booking name `str`.

Author(s)

Bobby Reardon

See Also

`count_str_list`, which is also used within `assign_metrics_list`.

Examples

```
A_list = ['']*60
```

```
A_list[8] = 'Joe Bloggs'
```

```
In[1]: count_str_list(A_list, 'Joe Bloggs')
```

```
Out[1]: 1
```

count_list

Description

`count_list` is used to count the number of empty (ie. ' ') seats in a given row of a given plane.

Usage

```
count_list(A, rowlen, i)
```

Arguments

A

A list representing the names assigned to each seat on the plane. This list is assumed to be in the same format as the seating table of the database, with entries ordered by seat letter, then by row number (ie, 1A, 2A, ... 15A, 1B, ...).

rowlen

Integer representing the number of seats in each row of the plane.

i

An integer representing the row of interest, where 0 represents the first row.

Details

`count_list` is used within `assign_metrics_list` to count the number of available seats in a given row by counting the number of empty ' ' entries. It applies the same rectangular plane assumption as the rest of the program.

Outputs

`count_list` returns the following outputs:

c

Integer representing the number of free/available seats remaining in row i.

Author(s)

Bobby Reardon

See Also

`count_str_list`, which is also used within `assign_metrics_list`.

Examples

```
A_list = [' ']*60
```

```
In[1]: count_list(A_list, 4, 2)
Out[1]: 4
```

assign_metrics_list

Description

`assign_metrics_list` is used to allocate seats, that are currently unoccupied on the plane, to a booking of given size and name.

Usage

```
assign_metrics_list(db, booking_name, booking_size, sep = 'Separations')
```

Arguments

`db`

A string representing the database file name where the details of the plane are stored, along with the current values of the metrics.

`booking_name`

A string representing the name used for the booking which will be use to designate the seats once the assigning is completed.

`booking_size`

An integer representing the size of the party to which seats will be assigned.s

`sep`

A string representing the interpretation of the `passengers_separated` metric to be used. This can take the inputs: `'Separations'` [default], `'Alone'`, `'Total'` and `'Dissatisfaction'`. See 'Details' for more.

Details

`assign_metrics_list` first accesses the database file `db`. It retrieves the dimensions of the plane from the `rows_cols` table and creates a list of names for each seat from the `seating` table.

A while loop is run to check if any prior bookings have the same name as the one being processed. Until a unique name is generated, the loop adds `' (1) '` to the end of the booking name so as to run the assign step correctly.

The assigning algorithm operates by trying to fit the largest possible group together in a single row, each time scanning rows from front to back. In the event that no row can accommodate the booking size, the booking is partitioned by separating a single individual and then attempting to assign seats to the slightly smaller booking size. The functions `count_list` and `count_str_list` (see below) are used in this process. The process iterates until it allocates seats to the largest possible subgroup of the original booking. Once completed it repeats the process if necessary for the remaining members of the party. At each step of the process, the seating table of the database file is updated.

Assigning step assumes all seats already booked have been done so in a left-to-right manner with no gaps between occupied seats within a given row.

A list of the row numbers of each occupied seat is then generated and this is then used in different manners to update the `passengers_separated` metric depending on the choice of interpretation.

Interpretations:

- `'Alone'` interprets the metric as the number of passengers seated who are completely separated from all other members of the party.

- `'Separated'` interprets the metric as the total number of party splits made when assigning the seats.

- `'Total'` interprets the metric as the total people sitting away from any other member of their party, ie. if a group is split, it is equal to the group size.

- `'Dissatisfaction'` interprets the metric by assigning dissatisfaction weightings to each booking which are defined as:

 - 0 if all party members are seated together.

 - 1 if party members are separated but no individuals are sat alone

 - 3 if one or more party members are sat alone.

Outputs

`assign_metrics_list` returns a printed statement `'Booking Confirmed'` once the database file has been updated correctly

Author(s)

Bobby Reardon

See Also

`count_list` and `count_str_list` functions used in `assign_metrics_list`.

Examples

```
import sqlite3
import csv
```

```
DB_file = "airline_seating.db"
booking_name = "Joe Bloggs"
booking_size = 6
```

```
assign_metrics_list(DB_file, booking_name, booking_size, 'Alone')
```

```
# Using default value of sep:
assign_metrics_list('plane.db', 'John Murphy', 4)
```


run_all

Description

run_all is used to run the entire program while allowing the user to specify an interpretation of the separations metric as well as providing the user control over which entries in the bookings file are run.

Usage

```
run_all(DB_file = 'airline_seating.db', CSV_file = 'bookings.csv', sep = 'Separated', first=1, last=0)
```

Arguments

DB_file

A string representing the database file name where the details of the plane are stored, along with the current values of the metrics. By default it will load the database file 'airline_seating.db' as per the sample file.

CSV_file

A string representing the bookings CSV file name where the ordered list of bookings are stored, with both booking names and corresponding sizes recorded. By default it will load the CSV file 'bookings.csv' as per the sample file.

sep

A string representing the interpretation of the passengers_separated metric to be used. This can take the inputs: 'Separations' [default], 'Alone', 'Total' and 'Dissatisfaction'. See 'Details' for more.

first

An integer representing the row number (1 indexed) of the bookings file for the algorithm to start at. By default, the algorithm begins with the first row.

last

An integer representing the last number (1 indexed) of the bookings file that the algorithm will run. By default, the algorithm ends with the last row.

Details

run_all begins by calling both the read_database and read_csv functions in order to extract the relevant information. It then sets the first and last rows of the CSV file to be processed.

For each of the k (end - start) bookings to be processed, the function begins by evaluating whether the plane can still accommodate the entire booking size. If it cannot be processed, the passengers_refused metric is updated and the new value is printed to screen. If it can be

processed, the `assign_metrics_list` function is called and the seating table and `passengers_separated` metric within the database are subsequently updated.

Outputs

`run_all` returns a printed statement for each booking which was accommodated and assigned along with the size of the booking. If the booking has failed to be assigned, the function prints a message to confirm no more free seats exist and prints the updated `passengers_refused` metric.

Author(s)

Bobby Reardon, Eoin Carroll

See Also

`run_all` comprises of three constituent functions, namely `readDB`, `readCSV` and `assign_metrics_list`. See these for more details.

Examples

```
import sqlite3
import csv
```

```
DB_file = "airline_seating.db"
CSV_file = "bookings.csv"
```

```
# Run entire bookings list with default metric interpretation:
run_all(DB_file, CSV_file)
```

```
# Or alternatively
run_all()
```

```
# Run first 10 entires using 'Alone' interpretation:
run_all(DB_file, CSV_file, 'Alone', 1, 10)
```

read_database

Description

read_database is a function that reads the seat layout from a database file to record the layout of the plane. It also checks to see how many free seats are available.

Usage

```
read_database(db_file = "airline_seating.db")
```

Arguments

db_file

The input argument represents the filename of the database to be used. It defaults to "airline_seating.db" for the purpose of this assignment.

Details

This function utilises the `sqlite3` library to interact with an SQL database. It opens the specified file and returns specific values from different tables in the database. It also iterates through the seat table and calculates the number of free seats.

We assume that the database is set up correctly as per the sample database provided for this assignment.

Outputs

read_database returns the following outputs:

number_of_rows

Represents the number of rows on the plane taken from the database.

seat_layout

Represents the seating format in the form of seat letters per row.

free_seats

Represents the number of unoccupied seats in the database.

Author

Eoin Carroll

Example

```
import sqlite3
```

```
DB_file = "airline_seating.db"
```

```
nrows, seat_layout, free_seats = read_database(DB_file)
```

read_csv

Description

`read_csv` is a function that reads the list of passengers that wish to be booked onto the plane from a CSV file.

Usage

```
read_csv(csv_file = "bookings.csv")
```

Arguments

`csv_file`

The input argument represents the filename of the CSV file to be used. It defaults to `"bookings.csv"` for the purpose of this assignment.

Details

This function utilises the CSV library to interact with the CSV file. It opens the specified file and reads the name and number for each passenger booking.

We assume that the CSV file is set up correctly as per the sample file provided for this assignment. The assumed format contains the passenger names in column 1 and the passenger group sizes in column 2.

Outputs

`read_csv` returns the following outputs:

`booking_number`

Represents the number of unique booking requests

`booking_name`

Represents the name of each booking. List format.

`booking_size`

Represents the size of each booking. List format.

Author

Eoin Carroll

Example

```
import csv
```

```
CSV_file = "bookings.csv"
```

```
booking_number, booking_name, booking_size = read_csv(CSV_file)
```