You are developing a taxi management system. The taxi drivers are the ones that use this system to process and print out receipts of the customers' trips. You are required to implement the program following the below requirements

There are 3 parts; please ensure you only attempt one part after completing the previous part.

**------------------------------------------------ PART 1 (7 marks) ------------------------------------------------**

In this part, your program can perform some simple interactions with users (i.e., the taxi drivers):

1. Display a message asking the user to enter the customer’s name. In this part, you can assume

a customer name only consists of alphabet characters.

2. Display a message asking the user to enter the departure location. In this part, you can assume

the location to be entered is always a valid location.

3. Display a message asking the user to enter the destination location. In this part, you can assume

the location to be entered is always a valid location and it needs to be different compared to the

departure location.

Page 3 of 12

RMIT Classification: Trusted

4. Display a message asking the distance (in km) between the departure and the destination

locations. In this part, you can assume the distance entered is always a positive valid number,

e.g., 10.2, 20.5.

5. Display a message asking the user to enter the rate type. There are 4 types of rates: *standard,*

*peak, weekends, holiday*. The prices per 1 km for these types are *1.5, 1.8, 2, 2.5*, respectively.

In this part, you can assume the rate type entered by the user is a valid value (within the list of

rate types).

6. Calculate the total cost for the customer including the discount. The total cost is equal to the

distance fee + basic fee – discount. The basic fee is always 4.2$. The distance fee is computed

based on the distance and the price per km, for example, if the distance is 10km, the rate is

standard, then the distance fee is 1.5x10 = 15$. For the discount, see No. 7 below.

7. An existing customer will have a discount of 10% over the distance fee (not apply to basic fee).

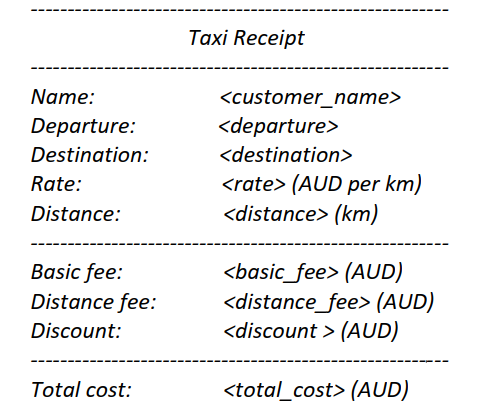
For example, if the distance fee is 15$, then the discount will be 1.5$. There is no discount for

new customers.

8. All the booking information will be displayed as a formatted message to the user as follows.

Note that the rate, distance, basic fee, distance fee, discount, and total cost are all displayed

with two digits after the decimal point.

  
9. In the program, you should have some lists (or dictionaries or other data types) to store the

names of all customers, the available locations, the available rate types, the prices of those rate

types. You can assume the customer names, locations, and the rate types are all unique and

case sensitive.

10. When a new customer finished booking a taxi trip, your program will automatically add the

customer's name to the customer list. When any customer books a trip, your program will check

if they are an existing customer. If yes, the discount fee is applied.

11. Your program needs to be initialized with the following existing customers: *Louis* and *Ella*,

and the following locations: *Melbourne, Chadstone, Clayton, Brighton, Fitzroy*.

12. Note: in the requirements No. 9 & 10, we use the term 'list' when describing the customer list,

the locations, etc, but you can use other data types to store this information such as dictionaries

and other data types. Make sure you think and analyse the requirements in detail so that you

can choose the most appropriate/suitable data types.

Page 4 of 12

RMIT Classification: Trusted

**---------- PART 2 (4 marks, please do not attempt this part before completing PART 1) -----------**

In this part, your program can: (a) perform some additional requirements compared to PART 1, and (b) be operated using a **menu**.

First, compared to the requirements in PART 1, now your program will be able to handle invalid inputs from users:

a. Display an error message if the customer’s name entered by the user contains non-alphabet

characters. When this error occurs, the user will be given another chance, until a valid name

(names contain only alphabet characters) is entered.

b. Display an error message if the location entered by the user is not a valid location. When this

error occurs, the user will be given another chance, until a valid location is entered. For the

departure location, a valid location is a location in the location list. For the destination

location, a valid location is a location in the location list and is different from the departure

location.

c. Display an error message if the rate type entered by the user is not a valid rate type (e.g.,

standard, peak, etc.). When this error occurs, the user will be given another chance, until a

valid rate type is entered.

d. Display an error message if the distance entered is 0, negative, or not a number. When this

error occurs, the user will be given another chance, until a valid distance is entered.

Second, your program will be operated using a **menu**. A menu-driven program is a computer program in which options are offered to the users via the menu. Your program will have the following options: book a trip, add/update rate types and prices, display existing customers, display existing locations, display existing rate types, and exit the program (please see Section 5 in this document regarding an example of how the menu program might look like). Below are the specifications of the options:

1. *Book a trip*: this option includes all the requirements from 1 to 12 in PART 1 and the

requirements a to d in the first part of PART 2.

2. *Add/update rate types and prices*: this option displays a message asking if users want to add

new or update existing rate types, and another message asking users to enter the prices of the

entered rate types. The rate types must be entered as a list that separates by commas, e.g.,

*metropolitan, deluxe, premium*. The prices will also be entered as a list separating by commas,

e.g., *2.0, 2.5, 2.2*. If the rate types are new, the rate types and their prices will be added to the

data collection of the program. If the rate types are the existing rate types, the newly entered

prices will replace existing prices. The order of the prices is the same as the order of the rate

types, e.g., in our example, the prices of metropolitan, deluxe, and premium are 2.0, 2.5, and

2.2 respectively. You can assume the rate types are always unique, and each rate type is a single

word with no comma but only alphabet characters. You can assume users always enter the

correct formats of the rate type and price lists, but note they can enter multiple spaces before

or after the commas. In this part, you can assume the prices entered are always valid and

positive numbers and has no comma.

3. *Display existing customers*: this option displays on screen all existing customers. The messages

are flexible (your choice).

4. *Display existing locations*: this option displays on screen all existing locations. The messages

are flexible (your choice).

5. *Display existing rate types*: this option displays all the rate types with their prices. The

messages are flexible (your choice), but they should show all the information required.

6. *Exit the program*: this option allows users to exit the program.

Page 5 of 12

RMIT Classification: Trusted

Note that in your program, when a task (option) is accomplished, the menu will appear again for the next task.

**----------- PART 3 (6 marks, please do not attempt this part before completing PART 2) ----------**

In this part, your menu program is equipped with some advanced features. Note, some features maybe very challenging.

1. In this part, in the *"Book a trip"* option, your program will allow customers to enter multiple

destinations. That is, after a destination and its corresponding distance to the departure location

is entered into the system, the user will be asked if another destination is added. If the answer

by the user is *y* (meaning yes), then the system will ask for the destination and the distance (in

relation to the previous destination). The process will be repeated until the user answers *n*

(meaning no). The cost of the trip will be based on the total distance entered. In this option,

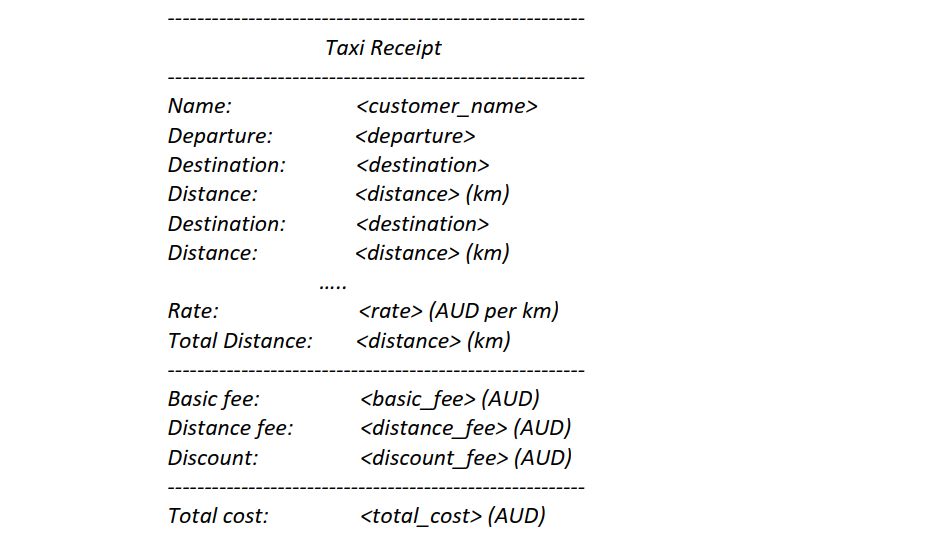
invalid inputs need to be handled. That is, the destination should be a valid destination and

should be different to the departure and the previous entered destination – if not, the user will

be given another chance until a valid location is entered. The answer by the user should only

be *y* or *n* – if the user enters a value that is different to *y* or *n*, they will be given another chance

until a valid answer is entered. The formatted message for the receipt is as follows.

2. In this part, your program will check the prices entered in the "*Add/update rate types and*

*prices*" option. If one of them is not a valid number, or negative number, or 0, the program will

ask the user to enter the price list again, until a valid list (contain all valid prices) is entered.

3. The menu now has an option *"Add new locations"* to add new locations to the program. This

option will accept a list of locations (separating by commas) as the input. For example, users

can enter the input: *Geelong, Carlton, Southbank* to add Geelong, Carlton, Southbank to the

location list of the program. If a location is new, then it will be added to the program. If a

location is an existing location, the program will display a message saying this is an existing

location so it will not do anything. You can assume the locations are always unique, and each

location is a single word with no space nor commas. You can assume users always enter the

Page 6 of 12

RMIT Classification: Trusted

correct formats (i.e., using commas to separate the locations), but note that users can enter multiple spaces before/after the commas.

4. The menu also has an option *"Display the most valuable customer*" to display the customer

with the maximum money (total cost) spent to date and the amount of money they have spent.

If there are multiple customers with the same maximum money spent, you can display only

one movie or all movies, it's your choice.

5. The menu now has an option *"Display a customer booking history".* This option will display a

message asking the user to enter the name of the customer, and the program will display all the

previous bookings of that customer, including the information of the departure and destination

locations of their bookings, and the total cost. Invalid customer names will be handled, and the

user will be given another chance until a valid customer name is entered. For example, if a

customer named Tom booked 2 previous trips, one trip from Melbourne to Chadstone then

Clayton with the total cost of 40.5$, and another trip from Docklands to Clayton with the total

cost of 35$, then the program will display the formatted message as follows.

*This is the booking history of Tom.*

|  |  |
| --- | --- |
| *Booking 1* | *Booking 2* |
| *Departure* | *Melbourne* | *Docklands* |
| *Destination* | *Chadstone, Clayton* | *Clayton* |
| *Total cost* | *40.5* | *35.0* |

A - Functionalities Requirements:

There are 4 levels, please ensure you only attempt one level after completing the previous level.  
PASS Level (10 marks) ------------------------------------------

At this level, your program will have some basic classes with specifications as below. You may need to define methods wherever appropriate to support these classes. At the end of the PASS level, your program should be able to run with a menu described in the class Operations.

Customers:

1. Class Customer

Each customer has a unique ID, unique name (a name will only contain alphabet characters). All existing customers are offered discount. You are required to write the class named Customer to support the following:

i. Attributes ID and name

ii. Constructor takes the values of ID, name as arguments

iii. Appropriate getter methods for the attributes of this class

iv. A method get\_discount which should be an empty super method

v. A method display\_info which should be an empty super method

2. Class BasicCustomer

All Basic customers have a flat discount rate when booking a trip if they have made a booking before. The class BasicCustomer should have the following components:

i. An attribute for the discount rate, by default it is 10%. Note that all Basic customers have the same discount rate.

ii. Constructor takes the appropriate parameters/arguments (be careful)

iii. Appropriate getter methods for the attributes of this class

iv. A method get\_discount which takes the distance fee and returns the discount. For example, this method returns 10 when the discount rate is 10% and the distance fee is 100$.

v. A method display\_info that prints the values of the BasicCustomer attributes.

vi. A method set\_discount\_rate to adjust the discount rate. This method affects all basic customers.

3. Class EnterpriseCustomer

Enterprise customers are customers from organizations that have a special deal with the taxi service provider. These customers are offered discount based on two rates: one when the distance fee is smaller than a threshold, and one when the distance fee larger than or equal to a threshold. Each Enterprise customer can have different rates, but the second rate is always 5% more than the first rate. The two rates are 15% and 20% if not specified. The threshold is the same for all Enterprise customers. The default threshold is 100$.

For example, if the threshold is 100$, and the two rates are 15% and 20%, then when an Enterprise customer books a trip with the distance fee being 60$ then the discount is 9$, and when they book a trip with the distance fee being 120$ then the discount is 24$.

The class EnterpriseCustomer should have the following components:

i. Appropriate attributes to support the two rates

ii. Constructor takes the appropriate parameters/arguments (be careful)

iii. Appropriate getter methods for the attributes of this class

iv. A method get\_discount which takes the distance fee and returns the discount offered

v. A method display\_info that prints the values of the EnterpriseCustomer attributes.

vi. A method set\_discount\_rate to adjust the discount rates of each individual Enterprise customer.

vii. A method set\_threshold to adjust the threshold limit. This affects all Enterprise customers.

Locations:

4. Class Location

This class is to keep track of information on different locations that the taxi service offers. This class supports the following information:

• ID: a unique identifier of the location

• name: the name of the location (you can assume the location names are unique and they do not include any digit)

• A method display\_info that prints the values of the Location attributes.

• Extra attributes and methods if you want to define.

Rates:

5. Class Rate

This class is to keep track of information on different rate types that the taxi service offers. This class supports the following information:

• ID: a unique identifier of the rate type

• name: the rate type (you can assume the rate types are unique and they do not include any digit)

• price: the price per km corresponding to the rate type

• A method display\_info that prints the values of the Rate attributes.

• Extra attributes and methods if you want to define.

Bookings

6. Class Booking

This class is to store a customer's booking information. This class supports the following information of a booking:

• customer: the one who makes the booking (can be a Basic or Enterprise customer). You need to think/analyse carefully if this should be an ID, name, or something else.

• departure: the departure the customer chooses. You need to think/analyse carefully if this should be an ID, name, or something else.

• destination: the destination the customer chooses. You need to think/analyse carefully if this should be an ID, name, or something else.

• distance: the distance of the trip chosen by the customer.

• rate: the rate type that the customer chooses. You need to think/analyse carefully if this should be an ID, name, or something else.

• A method compute\_cost that returns the distance fee, the basic fee, and the discount. For example, if the booking is of the customer Louis (Basic customer with discount rate 10%), the distance is 12, rate type is standard (price per 1 km: 1.5), then this method will return (18, 4.2, 1.8).

• Extra attributes and methods if you want to define.

Records

7. Class Records

This class is the central data repository of your program. It supports the following information:

• a list of existing customers – you need to think what you should store in this list (customer ID, customer name, or something else?)

• a list of existing locations – you need to think about what you should store in this list (location ID, location name, or something else?)

• a list of existing rate types that the taxi provider offers - you need to think about what you should store in this list (rate type ID, rate type’s name, or something else?)

• This class has a method named read\_customers. This method takes in a file name and then read and add the customers in this file to the customer list of the class. In the sequel, we call this the customer file. See an example of the customer file below.

In this file, the customers are always in this format: customer\_ID, customer\_name, customer type, discount\_rate (first discount rate for Enterprise customers), threshold (if that is an Enterprise customer). For example, in the 1st line, the customer\_ID is 1, the name is Louis, the customer type is B (Basic customer), the discount rate is 10%. In the 3rd line, the customer\_ID is 3, the name is George, the customer type is E (Enterprise customer), the discount is 15% (the first discount rate), and the threshold is 100. The IDs should be all unique. In this part, you can assume there will be no error in this customer file (e.g., the data format is always correct, and the discount values and thresholds are always valid).

• This class has another method named read\_locations. This method takes in a file name and can read and add the locations stored in that file to the location list of the class. In the sequel, we call this the location file. See an example of the location file below

In this file, the locations are always in this format: location\_ID, location\_name. The location ID always starts with the letter "L". The location IDs and names are all unique. You can assume there will be no error in this file (e.g., the data format is always correct, and the values are always valid).

• This class has another method named read\_rates. This method takes in a file name and can read and add the rates stored in that file to the rate list of the class. In the sequel, we call this the rate file. See an example of the rate file below.

In this file, the data are always in this format: rate\_ID, rate\_name, price\_per\_km. The rate ID always starts with the letter "R". The IDs and the names are all unique. You can assume there will be no error in this file (e.g., the data format is always correct, and the prices are always valid).

• This class also has three methods find\_customer, find\_location, and find\_rate. These methods take in a search value (can be either a name or an ID of a customer, location, or rate type), search through the list of customers/locations/rate types and then return the corresponding customer, location, rate type if found or return None if not found.

• This class also has three methods list\_customers, list\_locations, and list\_rates. These three methods can display the information of existing customers, locations, and rate types on screen. The method list\_customers will display the customer ID, name, customer type, the discount rate (first discount rate for Enterprise customers), and the threshold (for Enterprise customers). The method list\_locations will display the location ID and name. The method list\_rates will display the rate type ID, name, and the price per km. Note the three methods can be used to validate the reading from the three .txt files associated with the customers, locations, and rate types.

NOTE you are allowed to add extra attributes and methods in this class if these attributes and methods make your program more efficient.

Operations

8. Class Operations

This can be considered the main class of your program. It supports a menu with the following options:

i. Book a trip: this option allows users to book a trip for a customer. Detailed requirements for

this option are below (Requirements vi-ix).

ii. Display existing customers: this option can display all the information of all existing customers:

ID, name, customer type, discount rate (first discount rate for Enterprise customers), and

threshold (only for Enterprise customers).

iii. Display existing locations: this option can display all the information of all existing locations:

ID and name.

iv. Display existing rate types: this option can display all the information of all rate types: ID,

name, and the price per km.

v. Exit the program: this option allows users to exit the program.

Other requirements of the menu program are as follows:

vi. When the program starts, it looks for the files customers.txt (the customer file), locations.txt

(the location file), and rates.txt (the rate file) in the local directory (the directory that stores the .py file of the program). If found, the data will be read into the program accordingly, the

program will then display a menu with the 5 options described above. If any file is missing, the program will quit gracefully with an error message indicating the corresponding file is missing.

vii. Your menu program will allow the user to book a trip as specified in PART 1 of Assignment

1. In this part, like in PART 1 of Assignment 1, you can assume users always enter valid names,

locations, rate types, and distances.

viii. When a customer finishes booking a trip,

a. If the customer is a new customer, the program will add the information of that customer

into the data collection (think/analyse carefully which information needs to be added).

b. If the customer is an existing customer, the program will print out a message showing the

customer type (e.g., Basic/Enterprise customer), then proceed with the booking and display

the receipt.

ix. The receipt of a booking can be displayed as a formatted message as below.

---------------------------------------------------------

Taxi Receipt

---------------------------------------------------------

Name: <customer\_name>

Departure: <departure>

Destination: <destination>

Rate: <rate> (AUD per km)

Distance: <distance> (km)

---------------------------------------------------------

Basic fee: <basic\_fee> (AUD)

Distance fee: <distance\_fee> (AUD)

Discount: <discount > (AUD)

---------------------------------------------------------

Total cost: <total\_cost> (AUD)

x. When a task is accomplished, the menu will appear again for the next task. The program always exits gracefully from the menu.

Operations

At this level, your program needs to handle exceptions compared to the PASS level. At this level,you are required to define various custom exceptions to handle the below issues:

a. Display an error message if the customer’s name entered by the user contains non-alphabet characters. When this error occurs, the user will be given another chance, until a valid name (names contain only alphabet characters) is entered.

b. Display an error message if the location entered by the user is not a valid location. When this error occurs, the user will be given another chance, until a valid location is entered. For the departure location, a valid location is a location in the location list. For the destination location, a valid location is a location in the location list and is different from the departure location.

c. Display an error message if the rate type entered by the user is not a valid rate type (e.g., standard, peak, etc.). When this error occurs, the user will be given another chance, until a valid rate type is entered.

d. Display an error message if the distance entered is 0, negative, or not a number. When this error occurs, the user will be given another chance, until a valid distance is entered.

Besides, in this level, in the "Book a trip" option, your menu program has three more main features: (1) add extra service/package to the Book a trip option, (2) automatically load the services/packages when the program starts, and (3) a new option: Display the existing services/packages, and (4) support both names and IDs of the customers, locations, rate types and services. These features might be challenging. Details about these features are as below.

Firstly, in the option Book a trip: now your program will also display a message asking if the customer wants to order extra service/package (details about service/package are in the next paragraph). If yes, the program will display a message asking which service/package the customer wants to order. For each booking, the customer can only order one extra service/package. Note:

e. The discount rates of customers are only applied to the distance fee.

f. The total cost is equal the basic fee + distance fee – discount + service fee. For example, if the basic fee is 4.2, the distance fee is 15$, the discount is 1.5$, and the service fee is 2$, then the total cost is 4.2+15-1.5+2 = 19.7$.

g. Display an error message if the answer by the user is not y or n when asking if customer wants to order extra service/package. When this error occurs, the user will be given another chance, until a valid answer (i.e., y, n) is entered. You can assume the user always enters correct extra service name (Internet, Snack, Drink, Entertainment, etc.).

h. The receipt now includes the information of the service (service/package name and its price)

To support the above feature (i.e., order extra service/package), you need to add two more classes to your program and modify an existing class (note that you can modify more classes if you'd like).

9. Class Service

This class is to keep track of information of different extra services (Internet, Snack, Drink, Entertainment, etc.) that the taxi service provider supports. This class supports the following information:

• ID: A unique identifier for the service (e.g., S1, S2, etc.)

• Name: The name of the service (e.g., Internet, Snack, Drink, Entertainment, etc.).

You can assume the names of the services are unique. And you can assume the names can not include any digit.

• Price: The price of the service

The ID of a service always starts with letter "S". All the IDs and names are unique. You need to define the appropriate variables and methods to support the class Service. The price of a service can be changed. You are allowed to add extra attributes/methods if these attributes/methods make your program to be more efficient.

10. Class Package

This is a special kind of service. It means multiple services can be offered together as one service. For example, the Starter package consists of Internet and Entertainment. You can assume all components of a package are existing services in the system (i.e., the services are combined together to make a package).

The price of a package is 80% of the total price of all individual component services. For example, if Entertainment costs 3$, Internet costs 1$, Snack costs 2$, Drink costs 1$, then the price of the StarterMax package is 80% x (3 + 1 + 2 + 1) = 5.6$. Note that a package needs to have at least 2 services.

Each package has a unique ID and name (as with service). You need to define the appropriate variables and methods to support the class Package.

The class Records will then store more information:

• a list of existing services/packages

• an additional method named read\_service, which can read a comma separated file called services.txt and add services/packages to the service list. See an example of the services.txt file as below.

In this file, the services are always in this format: ID, name, and the *price of the service*. On the other hand, the data of a package consists of all the component services of the package. For example, the package *StarterMax* consists the services S1, S2, S3, S4. The ID of a service always start with the letter "S" whilst the ID of a package always starts with the letter "P". The IDs/ names of the services and packages are all unique. The data format of a package is different compared to a service. You can assume all the component services in a package are unique (no duplicates) and existing services. You can assume packages are always stored at the end of a file, after all normal services. You can again assume there are no errors in the file services.txt.• Additional methods named find\_service and list\_services to find a service (including  
packages) and list all existing services (including packages). The specifications are same as the find\_location and list\_locations methods.  
Secondly, when your program starts, it also looks for the file services.txt in the local directory (the directory that stores the .py file). If found, the data will be read into the program accordingly, the program will then display a menu. If any file (customer, location, rate, service) is missing, the program will quit gracefully with an error message indicating the files are missing.  
Thirdly, your menu program should now have an option Display existing services to list all the existing services/packages with the format as in the file services.txt (as specified in the Records class).  
Finally, your program should support both IDs and names of the customers, locations, rate types and services for any functions that use these information. For example, instead of entering the customer names, location names, rate type names, service/package names, now, the user can enter the customer IDs, location IDs, rate type IDs and service/package IDs, respectively.

**DI Level (3 marks, please do not attempt this level before completing the  
CREDIT level) -----------**In this level, there are some additional main features for some classes in your program. Some features  
might be challenging. Details of these features are described as follows.  
**Operations**Your program now has the following new features.  
1.In this level, in the "*Book a trip*" option, your program will allow customers to enter multiple destinations in one booking. The requirements are as in Assignment 1 for this option (requirement 1 of Part 3). You can modify the existing classes or design extra classes to support this requirement.

(requirement 1 of Part 3：In this part, in the *"Book a trip"* option, your program will allow customers to enter multiple destinations. That is, after a destination and its corresponding distance to the departure location is entered into the system, the user will be asked if another destination is added. If the answer by the user is *y* (meaning yes), then the system will ask for the destination and the distance (in relation to the previous destination). The process will be repeated until the user answers *n* (meaning no). The cost of the trip will be based on the total distance entered. In this option, invalid inputs need to be handled. That is, the destination should be a valid destination and should be different to the departure and the previous entered destination – if not, the user will be given another chance until a valid location is entered. The answer by the user should only be *y* or *n* – if the user enters a value that is different to *y* or *n*, they will be given another chance until a valid answer is entered.)

2.Your program now has an option "*Add new locations*" to add locations. The specification is same as Assignment 1 for this option (requirement 3 in Part 3).

(requirement 3 in Part 3：The menu now has an option *"Add new locations"* to add new locations to the program. This option will accept a list of locations (separating by commas) as the input. For example, users can enter the input: *Geelong, Carlton, Southbank* to add Geelong, Carlton, Southbank to the location list of the program. If a location is new, then it will be added to the program. If a location is an existing location, the program will display a message saying this is an existing location so it will not do anything. You can assume the locations are always unique, and each  
location is a single word with no space nor commas. You can assume users always enter thecorrect formats (i.e., using commas to separate the locations), but note that users can enter multiple spaces before/after the commas)

3.Your program now has an option "Adjust the discount rate of all Basic customers" to adjust the discount rate of all Basic customers. This adjustment will affect all Basic customers in all future orders. Invalid inputs (non-number or 0 or negative rate) should be handled via exceptions; the user will be given another chance until a valid input is entered.  
4. Your program now has an option *"Adjust the discount rate of an Enterprise customer"*. The option will ask for the name or ID of an Enterprise customer, then ask for a new first discount rate (e.g., 0.2 which corresponds to 20% discount rate). Invalid customers (non-existent or nonEnterprise customers) needs to be handled, i.e., your program will give the user another chance until a valid Enterprise customer is entered. Invalid inputs (non-number or 0 or negative values) should also be handled via exceptions, and the user will be given another chance until a valid input is entered. Note that your program should support both customers' IDs and names in this option, i.e., users can type either the customer's name or the ID.  
v. Note, in this part, you need to analyse the requirements and update some classes so that your program can satisfy the requirements listed above.

**HD level (6 marks, please do not attempt this level before completing  
the DI level) --------------**At this level, there are some additional features for some classes in your program. Note that some of  
them are very challenging (require you to optimize the class design and add components to support the features). Your program now will have the following features.  
1. The program now can load previous bookings via a file which are stored in a comma separated file named *bookings.txt* that is in the same directory with the .py file. This file will be loaded when the program starts (as with the customer, location, rate, and service files). In the sequel, we will call this the *booking file*. Below is an example of the booking file:  
Each line in the booking file is the information of a booking. The format is: *customer\_name/ID, departure\_name/ID, destination1\_name/ID, distance1, destination2\_name/ID, distance2, …, rate\_name/ID, service\_name/ID, basic\_fee, distance\_fee, discount, total\_cost.* You can assume all the customers in the booking file are existing customers (they are in the customer file). You can assume all locations in the booking file are existing locations (they are in the location file)  
and are valid. You can assume all rate types in the booking file are existing rate types (they are in the rate file). The services can be normal services or packages and they are all in the service file (when there is no service, there will be an empty field). Customers, locations, rate types, and services can be referred by IDs or names in this booking file. You can assume all other information (distance, basic fee, distance fee, discount, total cost) in this booking file is always valid, and the file entered by the user is always in the same directory with the .py file.  
Note that errors when loading the booking file should also be handled. When there are any errors loading the file, your program will print a message saying: "*Cannot load the booking file*".

2. Your program now has an option "Add/update rate types and prices" option to add/update rate types and prices. The specification is same as Assignment 1 for this option (requirement 2 in Part 2 and requirement 2 in Part 3). Note that for the invalid input handling, your program needs to use custom exceptions.  
(requirement 2 in Part 2 and requirement 2 in Part 3：

*Add/update rate types and prices*: this option displays a message asking if users want to add new or update existing rate types, and another message asking users to enter the prices of the entered rate types. The rate types must be entered as a list that separates by commas, e.g., *metropolitan, deluxe, premium*. The prices will also be entered as a list separating by commas, e.g., *2.0, 2.5, 2.2*. If the rate types are new, the rate types and their prices will be added to the data collection of the program. If the rate types are the existing rate types, the newly entered prices will replace existing prices. The order of the prices is the same as the order of the rate types, e.g., in our example, the prices of metropolitan, deluxe, and premium are 2.0, 2.5, and 2.2 respectively. You can assume the rate types are always unique, and each rate type is a single word with no comma but only alphabet characters. You can assume users always enter the correct formats of the rate type and price lists, but note they can enter multiple spaces before or after the commas. In this part, you can assume the prices entered are always valid and positive numbers and has no comma.  
In this part, your program will check the prices entered in the "*Add/update rate types and prices*" option. If one of them is not a valid number, or negative number, or 0, the program will ask the user to enter the price list again, until a valid list (contain all valid prices) is entered. )

3. Your program now has an option "*Display all bookings*" to display all bookings’ information as in the booking file (i.e., the customer's name, locations, distances, rate type, service, basic fee, distance fee, discount, total cost). The printed message is flexible

4. The program now can use command line arguments to accept five file names (the first being the customer file name, the second being the location file name, the third being the rate file name, the fourth being the service file, and the fifth being the booking file). The first fourth files are mandatory whilst the fifth file (booking) is optional. If no file names are provided, your program will look for customers.txt, locations.txt, rates.txt, services.txt and bookings.txt in the local directory. If a wrong number of arguments is provided, the program will display a message indicating the correct usage of arguments and exit

5. The menu now has an option "Display the most popular customer" to display the customer with the maximum money (total cost) spent to date and the amount of money they have spent. If there are multiple customers with the same maximum money purchased, you can display only

one customer or all customers, it's your choice.

6. Your program will now have an option "Display a customer booking history". The option will show a table displaying all the previous bookings of a particular customer. The specification is as in Assignment 1 but note it will include the information about the services/packages.

|  |
| --- |
| *This is the booking history of Louis.* |

|  |  |
| --- | --- |
| *Booking 1* | *Booking 2* |
| *Departure* | *Melbourne* | *Docklands* |
| *Destination* | *Chadstone, Clayton* | *Clayton* |
| *Service* | *Internet* | *Snack* |
| *Total cost* | *40.5* | *35.0* |

7. When your program terminates, it will update the four files: customers, locations, rates, and bookings, based on the information when the program executes.