Documentation

1. Function Specification

The program reads a list of airports with a given aircraft from a file and writes the most economic route to a another file. Before the list of airports is read, additional files are read to invoke representations of a given list of aircrafts and airports, each with their required data fields. Based on the specifications of the airports and the aircraft, the most economic route is found.

The cost of a single leg trip is the distance between the two airports multiplied with the to-Euro rate of the country the start airport is in.

Regarding the possible routes, the first airport in the list is assumed to be the start and end point of the journey, all other airports must be visited once and only once. To find the best route, all possible permutations of the remaining four airports are generated, to which the first airport is then added at the start and end. In an exhaustive loop through all possible routes, the total cost of each route is summed up and the route with the lowest total cost is returned.

If there are multiple routes that have the lowest total cost, the route that was encountered first is returned.

2. Design

Please see the UML diagram for more information on the classes.

Aircraft
Data field flightrange in kilometers: On Boing's website I found that they use nautical miles (nm) for their aircrafts, which has a conversion rate of 1.852 to kilometer. Our example solutions however are based on the regular mile with a conversion rate of 1.61 so I decided to assume regular mile as range unit.

Airport
The data fields needed for the calculation include the to-Euro rate. Once I realised that the rate value can only be obtained by mapping the airport and the rate via the country, adding it as a data field was straight forward.

Flightplan
A flightplan is a list of airports plus a given aircraft. This is the data structure at the center of the program: Its instances depend on the other classes airport and aircraft and its data fields can be manipulated in a particular way that is tailored to the problem. I decided to store the five airports of a flight plan in a Python list that

functions like a queue: New elements can be added at the start and the oldest element can be dequeued at the end (FIFO). Because only the first and last element need to the accessed for the purpose of this program, this seems like an efficient solution. Moreover, all possible routes of a flight plan can be retrieved with a method and are stored as tuples. A tuple is a immutable sequence type, which suits the purpose here as the routes do not need to be changed. The data fields best route and best cost are stored as tuple and as float respectively.

Due to time constraints, I was not able to implement graphs for the distances and costs between two airports, which would avoid repeated calculation of the same values.

AircraftList / AirportAtlas / FlightplanList
The purpose of each of these classes is to provide lookup options for the respective objects. This allows easy access and avoids duplicates. Additionally, an instance is only created if all required values are found in the files. For example, an airport is only added to the atlas if its rate, country, and currency code are found, otherwise it is ignored. Exception handling is included where appropriate, in particular for any I/O operations and key errors.

Complexity: Lookup O(1), Add O(1), Delete O(n)

pathfinder.py
Based on all possible routes and the aircraft's range, this script returns the best route and its cost. If the range is lower than a given distance, the route is ignored.

Complexity: Permutations factorial, Loop O(n)

run.py
Main script, provides a command line interface to the user. Checks validity of any required files before proceeding and prints status information to the console. Error handling with adequate messaging added to avoid unexpected stop of the program.

utils.py
Utility module, includes one function to write output to a given file. Looking back, it would have been worth refactoring some parts the program to reduce any duplicate code, for example the sections for reading input files.

directory.py
Small file to store paths of input files.

3. Testing: How have you tested your program.

Four test scripts can be run to exercise tests of the key components of the program. I developed the main classes in a test-driven approach from bottom up, which was

very helpful as it is a effective way of documenting tests and formalising the development process. However, the tests could cover more scenarios and should be written with proper testing syntax.

4. Completed features

1. Data structures for holding the information provided

2. Code to calculate the distance between airports

3. Code to price the cost of a single itinerary leg

4. A data structure to store an itinerary

5. Calculate the distance of each leg of a given route

6. Calculate the cost of each leg of a given route

7. Code to permute all of the possible itineraries

8. Code to price an given itinerary

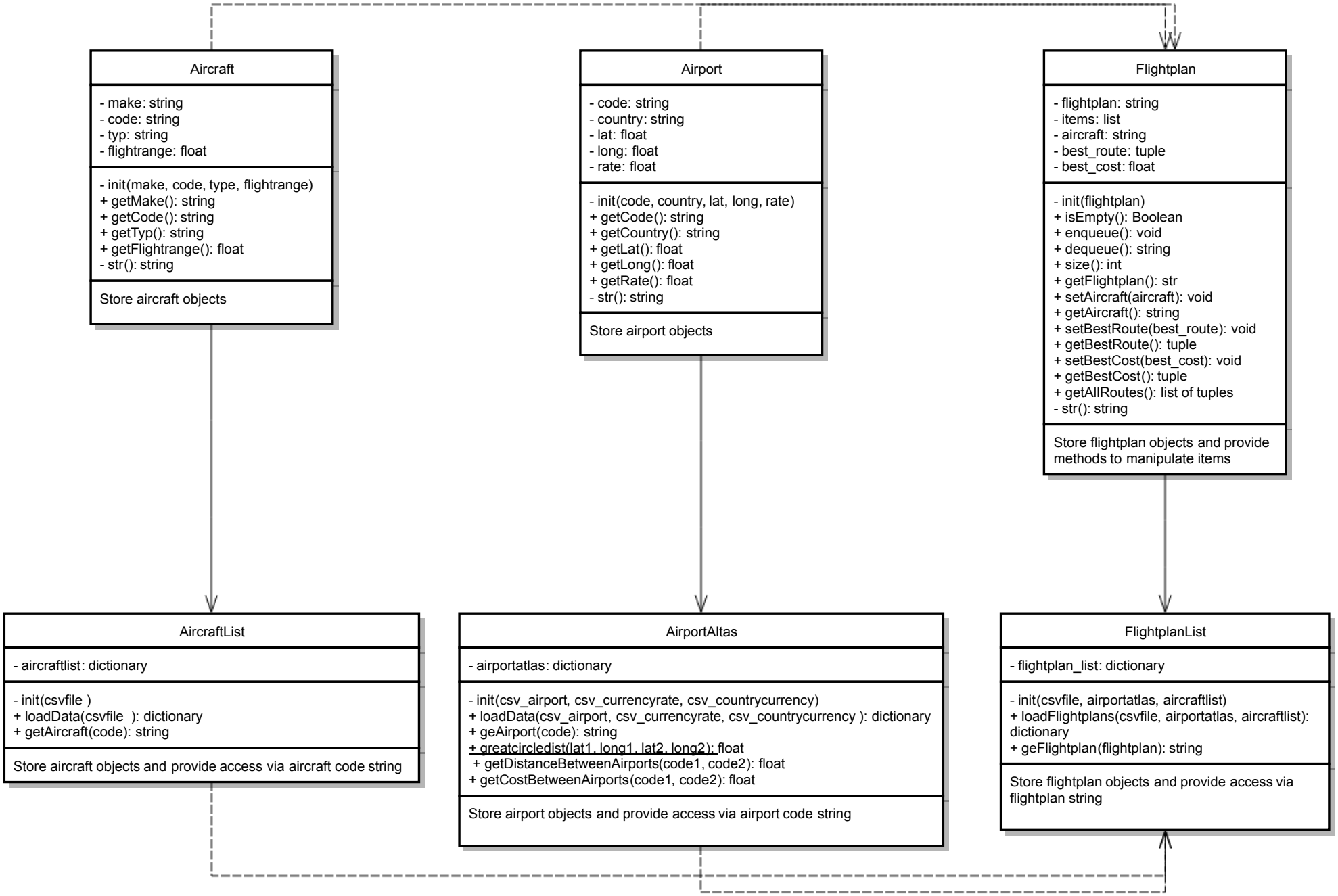
9. Calculate and display the cost of each leg of a given route

10. Calculate the best round trip route for an itinerary returning

11. Read in a csv as input and return a csv with the optimal route and cost

12. Basic unit tests

5. UML Diagram (see next page)


Aircraft
- make: string
- code: string
- typ: string
- flightrange: float
- init(make, code, type, flightrange)
+ getMake(): string
+ getCode(): string
+ getTyp(): string
+ getFlightrange(): float
- str(): string
Store aircraft objects
Airport
- code: string
- country: string
- lat: float
- long: float
- rate: float
- init(code, country, lat, long, rate)
+ getCode(): string
+ getCountry(): string
+ getLat(): float
+ getLong(): float
+ getRate(): float
- str(): string
Store airport objects
Flightplan
- flightplan: string
- items: list
- aircraft: string
- best_route: tuple
- best_cost: float
- init(flightplan)
+ isEmpty(): Boolean
+ enqueue(): void
+ dequeue(): string
+ size(): int
+ getFlightplan(): str
+ setAircraft(aircraft): void
+ getAircraft(): string
+ setBestRoute(best_route): void
+ getBestRoute(): tuple
+ setBestCost(best_cost): void
+ getBestCost(): tuple
+ getAllRoutes(): list of tuples
- str(): string
Store flightplan objects and provide methods to manipulate items
AircraftList
- aircraftlist: dictionary
- init(csvfile)
+ loadData(csvfile): dictionary
+ getAircraft(code): string
Store aircraft objects and provide access via aircraft code string
AirportAltas
- airportatlas: dictionary
- init(csv_airport, csv_currencyrate, csv_countrycurrency)
+ loadData(csv_airport, csv_currencyrate, csv_countrycurrency): dictionary
+ geAirport(code): string
+ greatcircledist(lat1, long1, lat2, long2): float
+ getDistanceBetweenAirports(code1, code2): float
+ getCostBetweenAirports(code1, code2): float
Store airport objects and provide access via airport code string
FlightplanList
- flightplan_list: dictionary
- init(csvfile, airportatlas, aircraftlist)
+ loadFlightplans(csvfile, airportatlas, aircraftlist): dictionary
+ geFlightplan(flightplan): string
Store flightplan objects and provide access via flightplan string