Introduction

Air travel is one of the most popular and efficient methods of transportation, especially for long-distance travel. Millions of people every year travel by airplane to visit family or friends. Air travel helps connect people around the globe. In this assignment, you are asked to implement a program that loads text files representing some of the airports and flights* around the world and analyzes the flights.

**The data in these files are not necessarily real/accurate.*

In this assignment, you will get practice with:

- Creating classes and objects
- Constructors, getters, setters, and other class methods
- Loading data from text files
- Cleaning and parsing data from text files
- Working with dictionaries and other data structures
- Algorithm development and testing; designing test cases
- Following program specifications

Text Files

In this assignment, there are 3 types of text files that will need to be read in.

The first is an airport file which lists a number of airports around the world. Each line in this type of file contains a single airport and it shows the 3-letter airport code, the country in which the airport is located, and then the city in which it is located. These three items are separated by hyphens. Some of them contain spaces and/or tabs around the individual portions of the line that must be cleaned up when being read in.

```
YYZ-Canada-Toronto
YVR-Canada-Vancouver
YHZ -  Canada-Halifax
YOW-Canada-  Ottawa
YEG-Canada-Edmonton
YUL-    Canada-Montreal
YWG-Canada- Winnipeg
 DFW -United States-Dallas
LAX-United States-Los Angeles
SFO-United States-  San Francisco
```

A small snippet showing the format of an airport file.

The second is a flight file that contains a list of flights including a 6-character flight code (3 letters and 3 digits with a hyphen between these two portions), the origin airport code, the destination airport code, and the flight duration in hours. These four items are separated by hyphens. Each line in the file represents one flight. Again, there may be spaces and/or tabs around the individual portions of the line that must be cleaned up when being read in.

```
XJX-595-LAX-CPT-19.27
 CSX-772- MAA-YHZ-23.48
LJC-201-FCO-YOW-12.54
EYS-649-YVR-PVG-12.40
OXD-016-ORD-    JFK  -   2.25
    DAJ-762-YOW-TIP-14.18
QUZ-869-YUL-MIA-4.01
RTK-498-YVR-LAX-2.95
VEB-477-PVG-PEK-2.45
SUF-706 -MAA-ICN- 7.15
```

A small snippet showing the format of a flights file.

The last is a maintenance file which lists the maintenance required to be done on some flights (overhaul, personnel, etc.). Each line in this file consists of a 6-character flight number (3 letters and 3 digits with a hyphen between these two portions), the airport in which the maintenance is to be done (irrelevant to the origin or destination of the flight), the hours of work needed to complete the maintenance, and the cost per hour of maintenance work. These four items are separated by hyphens. Similar to previous files, each line in the file represents one maintenance. Again, there may be spaces and/or tabs around the individual portions of the line that must be

cleaned up when being read in. Some lines might be empty, and some lines might contain identical content.

```
MQC-437 -MIA-50-12
NGF-735- YEG-65-10
   XUC-141-ATL-20 -15

QYR-830 -LAX-50-1
QYR-830 - LAX -50 -1
QYR-830 - LAX-50- 1
 XUC-141-ATL-20-15
QYR-830 - LAX-50- 1
EWQ-950-LAX-300-12
```

A small snippet showing the maintenance.txt file.

Python Files

For this assignment, you must create four (4) Python files: *Airport.py*, *Flight.py*, *MaintenanceRecord.py*, and *Assign4.py.*

When submitting your assignment on Gradescope, please submit these 4 files only. Do not upload any other files. **Make sure the files are named EXACTLY as specified here and all class, method, and function names match what is given EXACTLY including capitalization.**

Airport.py

The Airport file must contain a class called Airport. Everything in this file must be in the Airport class. Do not have any code in the file that isn't part of this class. As suggested by its name, this class represents an Airport in the program. Each Airport object must have a unique 3-letter code which serves as an ID, a city, and a country which are strings representing its geographical location.

Within the Airport class, you must implement the following functions based on these descriptions and specifications:

- **__init__(self, country, city, code):**
  - Initialize the instance variable *_code*, *_city*, and *_country* based on the corresponding parameters in the constructor

- **__str__(self):**
  - Return the string representation of this Airport in the following format:
    _code [*_city*, *_country*]
    i.e. YYZ [Toronto, Canada]
- **__eq__(self, other):**
  - Return True if self and other are considered the same Airport: if the 3-letter code is the same for both Airports.
  - If "other" variable is not an Airport object, it must also return False since Airport and non-Airport objects cannot be compared (hint: use the isinstance operator).
- **get_code(self):**
  - Getter that returns the Airport code
- **get_city(self):**
  - Getter that returns the Airport city
- **get_country(self):**
  - Getter that returns the Airport country
- **set_city(self, city):**
  - Setter that sets (updates) the Airport city
- **set_country(self, country):**
  - Setter that sets (updates) the Airport country

Flight.py

The *Flight* file must contain a class called *Flight*. Note that this file must import from *Airport* as it makes use of *Airport* objects; add the line `from Airport import *` to the top of the *Flight* file. Other than this import line, everything in this file must be in the *Flight* class. Do not have any other code in the file that isn't part of this class. As suggested by its name, this class represents a *Flight* from one *Airport* to another *Airport* in the program. Each *Flight* object must have a flight_no (a unique 6-character code containing 3 letters, hyphen, 3 digits), an origin airport, a destination airport, and a duration in hours. Both the origin and destination must be Airport objects, not strings, within the program.

Within the *Flight* class, you must implement the following functions based on these descriptions and specifications:

- **__init__(self, origin, destination, flight_number, duration):**
  - First check that both origin and destination are Airport objects (hint: use the isinstance operator). If either or both are not Airport objects, raise a TypeError that states "The origin and destination must be Airport objects".
  - If the origin and destination are both Airport objects, proceed to initialize the instance variable _flight_number, _origin, _destination, and _duration based on the corresponding parameters in the constructor.

- **__str__(self):**
  - Return the string representation of this *Flight* containing the origin city, destination city, duration (**rounded to the nearest int**), and an indication of whether the *Flight* is domestic or international (see the is_domestic function description below). The representation must be in the following format:
    origin_city to dest_city (domestic/international) [dur]
    Examples:
    Toronto to Montreal (domestic) [1h]
    Toronto to Chicago (international) [3h]
- **__eq__(self, other):**
  - Return True if self and other are considered the same *Flight*: if the origin and destination are the same for both *Flights*. If "other" variable is not a *Flight* object, it must also return False since *Flight* and non-*Flight* objects cannot be compared.
- **__add__(self, conn_flight):**
  - Check if conn_flight is a *Flight* object. If it is not, raise a TypeError to indicate that it cannot be added to a *Flight*. This exception should contain the text "The connecting_flight must be a *Flight* object".
  - If it is a Flight object, then check if the destination of the 'self' Flight is the same Airport as the origin of the 'conn_flight' Airport (i.e. one Flight ends and the next Flight begins in the same Airport). If not, raise a ValueError with the text "These flights cannot be combined" to indicate that these 2 Flight objects cannot be combined.
  - If they can be combined, then return a new Flight object with the origin set to self._origin, the destination set to conn_flight._destination, the duration set to the sum of the durations of both flights, and flight_no being equal to self._flight_no.
- **get_number(self):**
  - Getter that returns the Flight number code
- **get_origin(self):**
  - Getter that returns the Flight origin
- **get_destination(self):**
  - Getter that returns the Flight destination
- **get_duration(self):**
  - Getter that returns the Flight duration
- **is_domestic(self):**
  - Method that returns True if the flight is domestic, i.e. within the same country (the origin and destination are in the same country); returns False if the flight is international (the origin and destination are in different countries)
- **set_origin(self, origin):**
  - Setter that sets (updates) the Flight origin
- **set_destination(self, destination):**

- Setter that sets (updates) the Flight destination

MaintenanceRecord.py

The *MaintenanceRecord.py* file must contain a class called *MaintenanceRecord*. Do not have any other code in the file that isn't part of this class. As suggested by its name, this class represents a record from the *maintenance.txt* file. Each *MaintenanceRecord* object must have the following instance variables:

- *_flight*: of the *Flight* class type (and not a string),
- *_maintenance_airport:* of the type Airport (and not a string),
- _maintenance_duration in hours (a number), and
- *_maintenance_cost_per_hour* in dollars (a number).

This class receives a line from the maintenance.txt file in its constructor and processes the line according to the specification given below. Keep in mind that the MaintenanceRecord does not read the file directly, it is passed a string containing a line from the file.

Within the MaintenanceRecord class, you must implement the following methods based on these specifications:

- **__init__(self, input_line, all_flights, all_airports):**
  - Check the given *input_line* and extract the related parts. If it does not contain a flight number (3 letters followed by 3 digits, separated by a '-'), *maintenance_airport*, *maintenance_duration*, and *maintenance_cost_per_hour* (or lacks a combination of these) then raise a ValueError with the following message: "Invalid data string" (don't put any other information here).
  - Based on the flight number extracted from *input_line*, find the flight from within the *all_flights* dictionary and set the found flight as *self._flight*. If the flight is not in all_flights, raise a ValueError with the message "Flight not found".
  - Based on the maintenance airport code extracted from *input_line*, find the airport from *all_airports* and set the found airport as *self._airport*. If the airport is not in all_airports, raise a ValueError with the message "Airport not found".
  - Based on the given amounts for both duration of maintenance and cost per hour, set these variables as *self._maintenance_duration* and *self._maintenance_cost_per_hour*, respectively.
  - Example:
    - m1 = MaintenanceRecord("XUC-141-ATL-20-15", all_flights, all_airports) should result in the object m1 with following properties:
      - m1._flight = <Flight object for XUC-141 from JFK to DEN>
      - m1._maintenance_airpot = <Airport object with code ATL; Atlanta, Georgia>

        - m1._maintenance_duration = 20
        - m1._maintenance_cost_per_hour = 15
- **get_total_cost(self):**
    - Calculates the total cost of the maintenance and returns it. The total cost is the _maintenance_cost_per_hour * _maintenance_duration.
- **get_duration(self):**
    - Getter that returns the value of *self._maintenance_duration*.
- **__str__(self):**
    - Return the string representation of this MaintenanceRecord containing the flight code (from *self._flight* of type Flight), origin airport of the flight (of type Airport), maintenance airport (of type Airport), maintenance duration, hourly maintenance cost, and the total cost of the maintenance. The representation must be according to the following example:
      Examples, for lines "QYR-830 -LAX-50-1" and "NGF-735 -MIA-50-12", respectively:
    - QYR-830 (New York to Mexico City (international) [6h]) from JFK [New York, United States] to be maintained at LAX [Los Angeles, United States] for 50 hours @ $1.0/hour ($50.0)
    - NGF-735 (Dallas to New York (domestic) [4h]) from DFW [Dallas, United States] to be maintained at MIA [Miami, United States] for 50 hours @ $12.0/hour ($600.0)
- **__eq__(self, other):**
    - Return True if self and other are considered the same MaintenanceRecord: If all 4 properties (flight – of type Flight, maintenance airport – of type Airport, maintenance duration – number, and cost per hour – number) are equal, then the two MaintenanceRecord objects are the same. Otherwise, they are different. If other is not an instance of MaintenanceRecord, you should also return false.
        - Example:
            - m1 = MaintenanceRecord("XUC-141-ATL-20-15", all_flights, all_airports)
            - m2 = MaintenanceRecord("XUC-141-ATL-20-15", all_flights, all_airports)
            - m3 = MaintenanceRecord("XUC-141-ATL-20-50", all_flights, all_airports)
            - m1 == m2 should result in True
            - m1 == m3 should result in False
- **__ne__(self, other):**
    - Conversely to the __eq__ method, this method returns True if the self and other are not exactly the same. Returns False otherwise. It should also return True if other is not an instance of MaintenanceRecod.

- **__lt__(self, other):**
  - The "less than" method: returns True if the total cost of self (self.get_total_cost()) is less than total cost of other (other.get_total_cost()); False otherwise.
- **__le__(self, other):**
  - The "less than or equal to" method: returns True if the total cost of self (self.get_total_cost()) is less than or equal to total cost of other (other.get_total_cost()); False otherwise.
- **__gt__(self, other):**
  - The "greater than" method: returns True if the total cost of self (self.get_total_cost()) is greater than total cost of other (other.get_total_cost()); False otherwise.
- **__ge__(self, other):**
  - The "greater than or equal to" method: returns True if the total cost of self (self.get_total_cost()) is greater than or equal to total cost of other (other.get_total_cost()); False otherwise.

Assign4.py

This file is meant to be the core of the program from which the Airport and Flight objects should be created as their corresponding text files are loaded in; and several functions must be implemented to analyze the data and retrieve results about specific queries.

Before you start coding this file, see the template provided which may be helpful. It should be attached to the assignment on OWL.

Since this file will be used to create Airport and Flight objects, both of those Python files must be imported into this one. To do this, add the following lines of code to the top of this file:

```
from Flight import *
from Airport import *
from MaintenanceRecord import *
```

You then need to create three containers, **all_airports**, **all_flights**, and **maintenance_records** to store all the Airport, Flight objects, and maintenance records respectively. The **all_airports** container can be any type you wish to use that will work correctly, but the **all_flights** container **MUST** be a dictionary, and the **maintenance_records** container must be a list.

The rest of this file must be a series of functions based on these descriptions and specifications:

- **load_flight_files(airport_file, flight_file):**
  - Read in all the data from the airport file of the given name. Extract the information from each line and create an Airport object for each. Remove any whitespace from

the outside of **each portion** of the line (not just the line itself). As you create each Airport object, add the object to the all_airports container. Review the format of the airports file as well as the order in which the parameters are expected in the Airport constructor to ensure you send in the correct values for the correct parameters.

- Read in all the data from the flight file of the given name. Extract the information from each line and create a Flight object for each. Remove any whitespace from the outside of **each portion** of the line (not just the line itself). As you create each Flight object, add the object to the all_flights **dictionary** in this specific way: the key must be the origin's airport code and the corresponding value must be a list of the Flight object(s) that depart from this origin airport. For example, if "YYZ" is an entry in this dictionary, its value would be a list of all Flight objects in which the Flight's origin is "YYZ" (all flights that go out from Pearson airport). Review the format of the flights file as well as the order in which the parameters are expected in the Flight constructor to ensure you send in the correct values for the correct parameters.
- If both files load properly without errors, return True. If there is any kind of exception that occurs while trying to open and read these files, return False. Use a try-except statement to handle these cases.

- **get_airport_using_code(code):**
  - Return the Airport object that has the given code (i.e. YYZ should return the Airport object for the YYZ (Pearson) airport). If there is no Airport found for the given code, raise a ValueError with the text "No airport with the given code: CODE" where CODE should be replaced with the value of code such as "No airport with the given code: YYZ".
- **find_all_flights_city(city):**
  - Return a list that contains all Flight objects that involve the given city either as the origin or the destination. If there are no such flights, return an empty list.
- **find_all_flights_country(country):**
  - Return a list that contains all Flight objects that involve the given country either as the origin or the destination (or both). If there are no such flights, return an empty list.
- **has_flight_between(orig_airport, dest_airport):**
  - Check if there is a direct flight from origAirport to destAirport (both Airport objects). If so, return a True, otherwise return False.
- **shortest_flight_from(orig_airport):**
  - Determine the shortest flight in terms of duration in hours (**not rounded**) that originate from the given orig_airport Airport object. Return the corresponding Flight object that has the shortest duration. If there are 2 or more Flights tied for

the shortest from the given airport, return the first one that is stored in the dictionary (which should be the flight that occurs first in the flights file).

- **find_return_flight(first_flight):**
  - Take the given Flight object and look for the Flight object representing the return flight from that given flight. In other words, given a Flight from origin A to destination B, find the first Flight object that departs from origin B and arrives in destination A.
  - If there is no such Flight object that goes in the opposite direction as first_flight, raise a ValueError with the text "There is no flight from DEST_CODE to ORIG_CODE" where DEST_CODE and ORIG_CODE are the codes for the origin airport and destination airport for the first_flight object. For example: "There is no flight from ORD to YYZ".
- **create_maintenance_records(maintenance_file, flights_dict, airports_list):**
  - This method should read the contents of the maintenance_file and for each line create a MaintenanceRecord and add it to the maintenance_records list, but only if the list does not already contain an identical MaintenanceRecord. No duplicates should exist in this list. Keep in mind that some lines within the file might be empty.
  - If a line in the file is invalid (i.e. if the construction of MaintenanceRecord raises a ValueError) this function should return False. Otherwise, if all lines are valid this function should return True.
- **find_total_cost(records):**
  - This function returns the total cost of all maintenance records contained in the records list passed to this function. The records list is a list of maintenance records (note that this may not be the same list as maintenance_records).
  - It should iterate through the list and find the total maintenance cost of all maintenance records in the list and return the total cost.
- **find_total_duratation(records):**
  - This function returns the total hours that maintenance will take on all records contained in the records list passed to this function. The records list is a list of maintenance records (note that this may not be the same list as maintenance_records).
  - It should iterate through the list and find the total maintenance duration of all maintenance records in the list and return the total duration.
- **sort_maintenance_records(records):**
  - This function returns the given list of maintenance records in an order sorted by total cost from lowest cost to highest cost. You can use the built-in *sorted* function to implement this but note that the sorted function requires that the comparison methods (__lt__, __le__, __gt__, __ge__, __eq__, and __ne__) for the

MaintenanceRecord are created and working correctly to sort the objects in the correct order (see the description of the MainteanceRecord class).

- Example
  - Maintenance file:
  - RTK-498 -MIA-50-12
  - YOI-104- YWG-65-10
  - QYR-830 -LAX-50-1

```
sorted_list = sort_maintenance_records(maintenance_records)
print(sorted_list[0].get_total_cost(), sorted_list[1].get_total_cost(),
sorted_list[2].get_total_cost())
```

Expected result:

50.0 600.0 650.0

Code Template

The file Assign4.py attached to this assignment on OWL gives you a basic template for Assign4.py. It is strongly recommended that you use this template in your solution to ensure it works correctly with the autograder.

You should not have any code outside of the functions other than your declarations for **all_airports**, **all_flights,** and **maintenance_records**. Any code used to test your program should go inside of the **if __name__ == "__main__":** block at the bottom of the template so it does not impact the autograder.

Your files Flight.py, Airport.py and MaintenanceRecord.py should only contain a single class definition. **No code should be outside of the classes in these files.**

Test Case Examples

This section includes examples of function calls that you are encouraged to try and check if your output matches the expected output provided here. There are a limited number of examples here so you should also run your own tests beyond what is shown here. All examples here are based on the provided text files airports.txt and flights.txt.

load_flight_files(airport_file, flight_file)

```
data = load_flight_files("airports.txt", "flights.txt")
print(data, len(all_airports), len(all_flights))
```

Expected Output:

True 25 15

Note that the length of all_flights is 15 and not 25 even though the flights.txt file has 25 flights listed. Think about why it is only 15. Hint: remember what all_flights is and how it is storing the flights. Print out all items in all_flights if you are confused.

get_airport_using_code(code)

```
print(get_airport_using_code("ORD"))
```

Expected Output:

ORD [United States, Chicago]

```
print(get_airport_using_code("ABC"))
```

Expected Output:

ValueError: No airport with the given code: ABC

find_all_flights_city(city)

```
res = find_all_flights_city("Dallas")
for r in res:
    print(r)
```

Expected Output:

Dallas to New York (domestic) [4h]
Detroit to Dallas (domestic) [4h]

find_all_flights_country(country)

```
res = find_all_flights_country("China")
for r in res:
    print(r)
```

Expected Output:

Vancouver to Shanghai (international) [12h]
Shanghai to Beijing (domestic) [2h]
Beijing to Miami (international) [17h]

has_flight_between(orig_airport, dest_airport)

```
pearson = get_airport_using_code("YYZ")
ohare = get_airport_using_code("ORD")
edm = get_airport_using_code("YEG")
print(has_flight_between(edm, ohare))
```

Expected Output:

True

```
print(has_flight_between(edm, pearson))
```

Expected Output:

False

```
print(find_flight_between(pearson, ohare))
```

Expected Output:

False

shortest_flight_from(orig_airport)

```
jfk = get_airport_using_code("JFK")
print(shortest_flight_from(jfk))
```

Expected Output:

New York to Denver (domestic) [5h]

find_return_flight(flight)

```
sf_to_sp = all_flights["SFO"][0]
print(find_return_flight(sf_to_sp))
```

Expected Output:

Sao Paulo to San Francisco (international) [12h]

__add__(self, conn_flight) [in Flight.py]

```
f1 = all_flights["YEG"][0]
f2 = all_flights["ORD"][0]
print(f1 + f2)
```

Expected Output:

Edmonton to New York (international) [6h]

```
print(f2 + f1)
```

Expected Output:

ValueError: These flights cannot be combined

create_maintenance_records("file.txt", all_flights, all_airports)

with a file.txt like the following and the previous flights and airports (notice the blank spaces):

“ QYR-830 -LAX-50-1

XUC-141-ATL-20 -15

QYR-830 - LAX -50 -1

QYR-830 - LAX-50- 1

XUC-141-ATL-20 -15 ”

```
create_maintenance_records("file.txt", all_flights, all_airports)
print(len(maintenance_records))
print(maintenance_records[0])
print(maintenance_records[1])
```

Expected Output:

2

QYR-830 (New York to Mexico City (international) [6h]) from JFK [New York, United States] to be maintained at LAX [Los Angeles, United States] for 50 hours @ $1.0/hour ($50.0)

XUC-141 (New York to Denver (domestic) [5h]) from JFK [New York, United States] to be maintained at ATL [Atlanta, United States] for 20 hours @ $15.0/hour ($300.0)

find_total_cost(records)

```
m1 = MaintenanceRecord("YOI-104-ATL-1-2", all_flights, all_airports)
m2 = MaintenanceRecord("RTK-498-ATL-15-5", all_flights, all_airports)
m3 = MaintenanceRecord("ADJ-602-ATL-100-10", all_flights, all_airports)
print(find_total_cost([m1, m2, m3]))
```

Expected Output:

1077.0

find_total_duration(records)

```
print(find_total_duration([m1, m2, m3]))
```

Expected Output:

116

sort_maintenance_records(records)

```
recs = [m1, m2, m3]
print(recs[0].get_total_cost(),
      recs[1].get_total_cost(),
      recs[2].get_total_cost())
sorted_recs = sort_maintenance_records(recs)
print(sorted_recs[0].get_total_cost(),
      sorted_recs[1].get_total_cost(),
      sorted_recs[2].get_total_cost())
```

Expected Output:

1000.0 75.0 2.0
2.0 75.0 1000.0

Tips and Guidelines

- Variables should be named in lowercase for single words and snake case for multiple words, i.e. orig_airport.

- Do not hardcode filenames, numbers of lines from the files, or any other variables. Filenames may be different on the autograder. Always assume files are in the current working directory (i.e. only give the filename to open() and not the full absolute path).
- Make sure your class, method, and function names match those given in this document EXACTLY including capitalization. The autograder will not be able to find your classes or methods if you name them differently.
- Add comments throughout your code to explain what each method or significant block of code is doing and/or how it works.

Rules

- Read and follow the instructions carefully.
- You may **not** access instance variable directly outside of the class they belong to, all access must go through the getter and setter methods. Directly accessing instance variable outside of the class they belong to may lead to a mark deduction.
- Submit all the Python files described in the Files section of this document and no additional files.
- Submit the assignment on time. Late submissions will not be accepted or will be marked as 0, except where late coupons are used (see the course syllabus for details).
- Forgetting to submit a finished assignment is **not** a valid excuse for missing a due date.
- Make regular backups of your work and store it in multiple places, i.e. USB key, on the cloud, etc. Experiencing technical issues resulting in lost or corrupted files is **not** a valid excuse for missing a due date.
- Submissions must be done on Gradescope. They will **NOT** (under any circumstances) be accepted by email.
- You may re-submit your code as many times as you would like. Gradescope uses your last submission as the one for grading by default. There are no penalties for re-submitting. However, re-submissions that come in after the due date **will** be considered late and marked as 0, unless you have enough late coupons remaining, in which case late coupons will be applied.
- The top of **each** file you create must include your name, your Western ID, the course name, the date on which you **created** the file (no need to change it each time you modify your code), as well as a description of what the file does. This information is **required**. You must follow this format:

```
"""
*****************************
CS 1026 - Assignment X – Interest Calculator
Code by: Jane Doe
```

```
Student ID: jdoe123
File created: March 01, 2025
*****************************
This file is used to calculate monthly principal and interest amounts for a given
mortgage total. It must calculate these values over X years using projected variations in
interest rates. The final function prints out all the results in a structured table.
"""
```

The description should be updated for each file the comment is in and describe the contents of the file.

- Close all files you open.
- Assignments will be run through a similarity checking software to check for code that looks very similar to that of other students. Sharing or copying code in any way is considered plagiarism and may result in a mark of zero (0) on the assignment and/or reported to the Dean's Office. Plagiarism is a serious offence. Work is to be done **individually**.

What constitutes plagiarism

Plagiarism, or academic dishonesty, comes in a variety of forms including:

- sending any portion of your code to peer(s)
- letting peer(s) look at your code
- using any portion of code from peer(s)
- using ChatGPT or other AI platforms to generate any portion of code or comments
- using code that you found online
- paying someone to do your work
- being paid to do someone else's work
- leaving your work on a computer unattended where peers may be able to find it
- uploading your code to an online repository where peers may be able to access it
- allowing anyone else to use your Gradescope account for any reason

Submission

Due: Friday, April 3rd, 2025 at 11:59PM

You must submit the 4 files (Airport.py, Flight.py, MaintenanceRecord.py, and Assign4.py) to the Assignment 4 submission page on Gradescope. There are several tests that will automatically run when you upload your files. You will see the score of the tests but you will not see what is being tested. It is recommended that you create your own test cases to check that the code is working properly in many different scenarios.

Assignments will not be accepted by email or any other form. They **must** submitted on Gradescope.

Marking Guidelines

The assignment will be marked as a combination of your auto-graded tests and manual grading of your comments, formatting, etc. Below is a breakdown of the marks for this assignment:

[20 marks] Auto-graded tests.

[1 mark] Header section with name, student ID, course info, creation date, and description of file.

[1 marks] Comments throughout code including documenting methods (return, parameters, description).

[1 marks] Meaningful variable names.

[2 marks] Good programming practices *(e.g. do not access instance variable directly outside of the class they belong to, close all files you open, no code outside of classes/functions other than given in template, other rules given in assignment document)*

Total: 25 marks

IMPORTANT:

A mark of zero (0) will be given for tests that do not run on Gradescope. It is your responsibility to ensure that your code runs on Gradescope (not just on your computer). Your files must be named correctly, and you must follow all instructions carefully to ensure that everything runs correctly on Gradescope.

A mark of zero (0) will be given for hardcoding results or other attempts to fool the autograder. Your code must work for **any** test cases.

Directly accessing instance variable outside of the class, they belong to may lead to a manual mark deduction. Use getters and setters to access these outside of the class.

The weight of this assignment is 8% of the course mark.