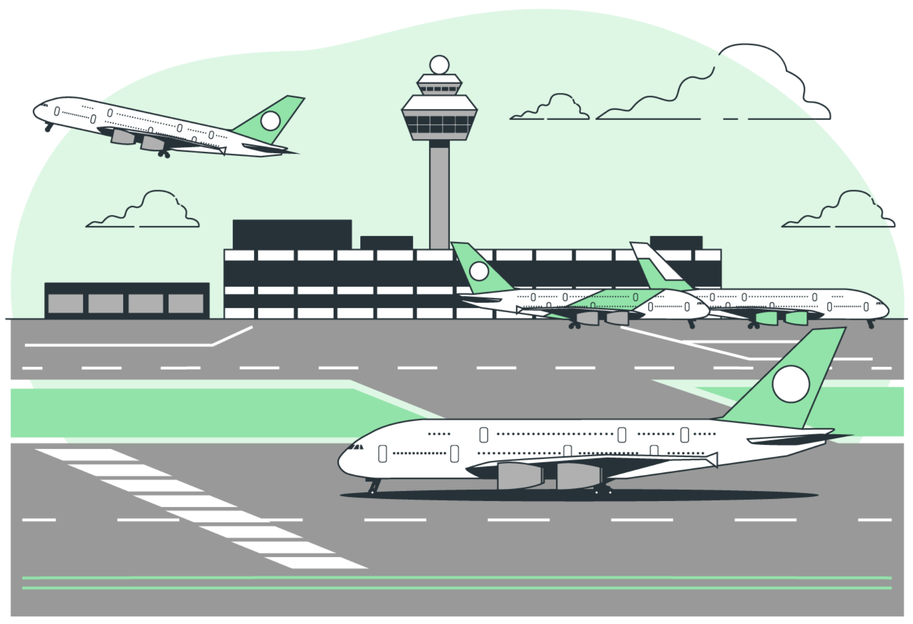
Western Science

CS 1026 : Computer Science Fundamentals



ASSIGNMENT 04

AVIATION SYSTEM

Due Date:
Wednesday April 5th, 2023 at 11:55 PM EST

-164d -20h -32m -3s left

The following are the instructions for assignment 04:

TOP

If you want to store a PDF version of this assignment, press Ctrl+p *on Windows or* Command+p *on Mac, the print window withh appear. Then, select* Save As PDF *from the* **Destination** *dropdown. Then click* Save

- INTRODUCTION

In this assignment, you will get practice with:

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

Air travel is one of the most popular, efficient, and safe methods of transportation, especially for long-distance travels. As we approach the summer season, we can expect that millions will be travelling by airplanes to visit family or friends. Airlins and air travel agencies helps connect people around the globe using aviation management systems. 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 according to specifications.

> Some of the data in these files are not necessarily real/accurate.

TOP

Here is a wlakthrough video that explains the assignment and the testing procedure:

1 **FILES**

In this assignment, there are 3 types of text files that will need to be read in using your code. **These three files are already provided to you**. When reading the file please consider:

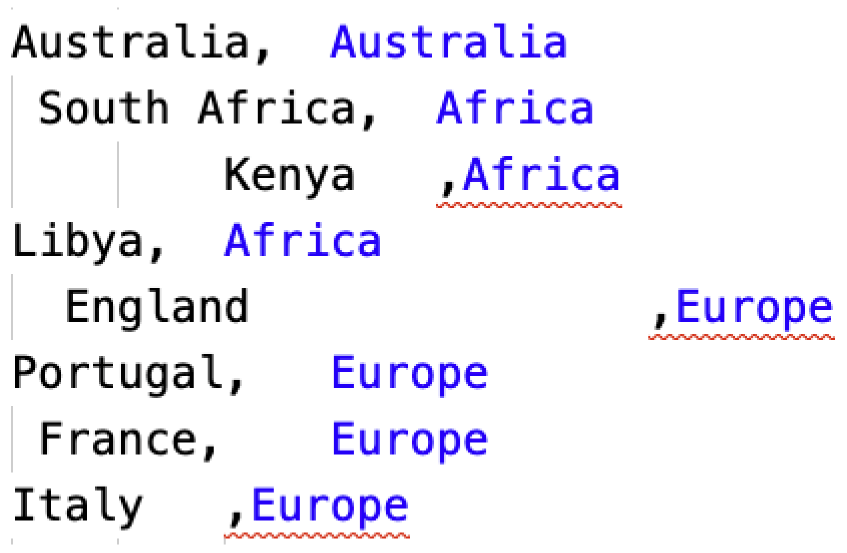
- You can assume that the data in these three files is **case-insensitive**.
- The data is separated by commas (,).
- You don't need to validate the data. You can assume that the data is correct. However,
- The data is not cleaned. You need to clean the data from extra white spaces and tabs.
- Empty lines should be ignored.

countries.txt

This file contains a set of lines with comma separated values. Each line contains a **country name** and the **continent name** that the country belongs to. Some of the lines contain spaces and/or tabs around the individual portions of the line that must be cleaned up when being read in. **You can assume that all the lines are separated correctly with commans and no other delimiter is used.**

A small snippet showing the format of an countries file:

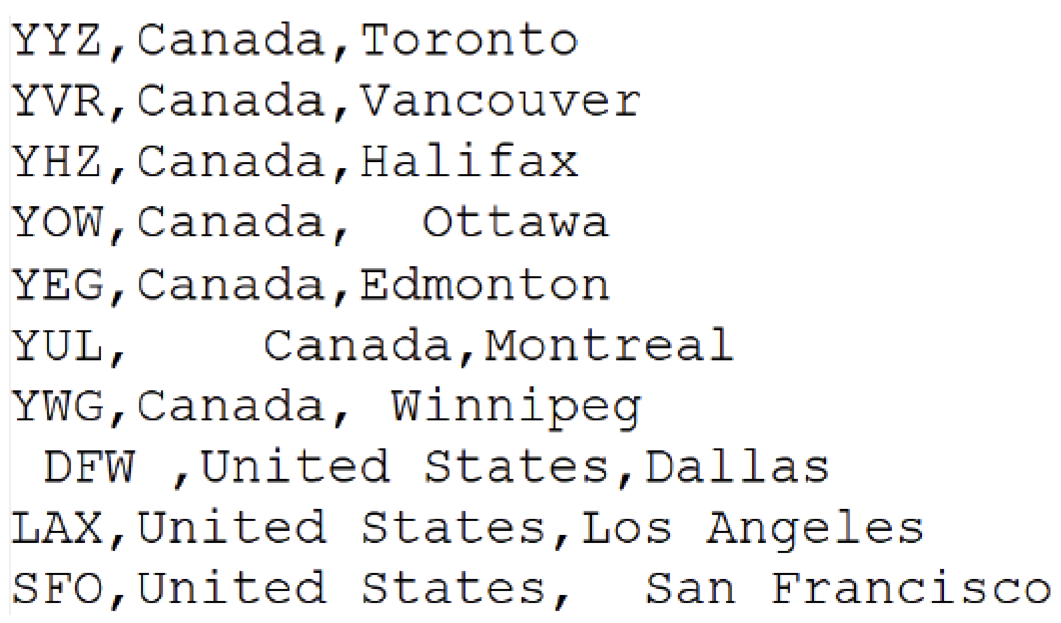
TOP


```
Australia,  Australia
 South Africa,  Africa
        Kenya   ,Africa
Libya,  Africa
  England                ,Europe
Portugal,   Europe
 France,    Europe
Italy   ,Europe
```

airports.txt

This is the **airports file** which lists a number of airports around the world. Each line in this 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 the airport is located. These three items are separated by commas. Some of the lines contain spaces and/or tabs around the individual portions of the line that must be cleaned up when being read in. **You can assume that all the lines are separated correctly with commans and no other delimiter is used.**

A small snippet showing the format of an airport file:


```
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
```

flights.txt

The other file is the **flights file** that contains a list of flights including a 6-character **flight code**, the **origin airport code**, and the **destination airport code**. 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. **Again, you can assume that all the lines are separated correctly with commans and no other delimiter is used.**

A small snippet showing the format of a flights file:

TOP

```
XJX595,LAX,CPT
 CSX772, MAA,YHZ
LJC201,FCO,YOW
EYS649,YVR,PVG
OXD016,ORD, JFK
    DAJ762,YOW,TIP
QUZ869,YUL,MIA
RTK498,YVR,LAX
VEB477,PVG,PEK
```

For this assignment, you must create three (3) Python files: `Airport.py`, `Flight.py`, and `Aviation.py`.

When submitting your assignment on Gradescope, please submit these 3 files only. Do not upload any other files.

Make sure the files are named EXACTLY as specified here.

REMEMBER

You **MUST** implement all classes, methods (with parameters names), functions listed in the description below, with **THE EXACT SAME NAMES (case sensitive)**. Failure to comply with this may result into a potential loss of marks. However, you are welcome to add more helper functions/methods as long as you you implement the required ones.

Please, adhere with the **EXACT** formatting of the outputs and return values. Pay attention to the punctuation used, paranthesis, brackets. "It is only a comma" is not a valied execuse :). Extra space here or there is fine.

Airport.py

The Airport file must contain a class called **Airport**. Every method in this file must be in the Airport class. Do not have any code in the file that isn't part of this class. **(Do not include functions calls, a main function, input(), etc. as this may result of the autograder to fail. This is your responsibility**).

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, country and a continent which all are strings representing its geographical location.

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

1. **__init__(self, code, city, country, continent):**
   Initialize the instance variables _code, _city, _country, and _continent based on the corresponding parameters in the constructor

TOP

2 **__repr__(self)**

Return the representation of this Airport in the following format:

```
{code} ({city}, {country})
```

e.g., YYZ (Toronto, Canada)

3 **getCode(self)**

Getter that returns the Airport code

4 **getCity(self)**

Getter that returns the Airport city

5 **getCountry(self)**

Getter that returns the Airport country

6 **getContinent(self)**

Getter that returns the Airport continent

7 **setCity(self, city)**

Setter that sets (updates) the Airport city

8 **setCountry(self,country)**

Setter that sets (updates) the Airport country

9 **setContinent(self,continent)**

Setter that sets (updates) the Airport continent

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. Therefore, you need to add the follwoing line to the top of the Flight file.:
from Airport import *
Other than this import line, everything in this file must be in the Flight class. Do not write any other code in the file that isn't part of this class. (No input, no function calls outside the class, etc. Again this is your responsibility :))
As suggested by the class name, this class represents a Flight from one Airport to another Airport in the program. Each Flight object must have a flightNo (a unique 6-character code containing 3 letters followed by 3 digits) as a string, an origin airport, and a destination airport. Both the origin and destination must be **Airport objects** within the program.

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

1 **__init__(self, flightNo, origAirport, destAirport)**

- First check that both origAirport and destAirport are Airport objects (hint: use the isinstance function). If either or both are not Airport objects, raise a TypeError exception that states:

```
"The origin and destination must be Airport objects"
```

TOP

- In the case that the flightNo is not a string of 6-character code containing 3 letters followed by 3 digits, raise a TypeError exception that states:
  `"The flight number format is incorrect"`
- When the origAirport and destAirport are both Airport objects, proceed to initialize the instance variables _flightNo, _origin, and _destination based on the corresponding parameters in the constructor.

2 **__repr__(self)**

Return the representation of this Flight containing the flightNo, origin city, and destination city, and an indication of whether the Flight is domestic or international (see the isDomesticFlight method description below). The representation must be in the following format:

```
Flight({flightNo}): {originCity} -> {destinationCity}
{[domestic]/[international]}
```

Examples:

Flight(MCK533): Toronto -> Montreal [domestic]

Flight(WCL282): Toronto -> Chicago [international]

3 **__eq__(self, other)**

Method that returns **True** if self and other flights are considered the same flight. Two flights are considered the same if the origin and destination are the same for both Flights i.e., self and other. Make sure that if "other" variable is not a Flight object, this means **False** should be returned.

4 **getFlightNumber(self)**

Getter that returns the Flight number string code

5 **getOrigin(self)**

Getter that returns the object of the Flight origin

6 **getDestination(self)**

Getter that returns the object of the Flight destination

7 **isDomesticFlight(self)**

This method returns True if the flight is domestic, i.e. within the same country (the origin and destination are in the same country); However, it returns False if the flight is international (the origin and destination are in different countries).

8 **setOrigin(self, origin)**

Setter that sets (updates) the Flight origin

9 **setDestination(self, destination)**

Setter that sets (updates) the Flight destination

TOP

Aviation.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 functionalities must be implemented to analyze the data and retrieve results about specific queries.

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 *
You need to create the class **Aviation** as the name suggests. Apart from the aforementioned two import lines, all the content of this file should relate only to the Aviation class. (No input, no function calls outside the class, etc. Again this is your responsibility :)). The following methods needs to be implemented based on these descriptions and specifications:

1. **__init__(self)**
   The constructor will not accept any external parameter. However, you then need to create several instance variables. Three containers, self._allAirports, self._allFlights, and self._allCountries to store all the Airport objects, Flight objects, and counrties with continents respectively. The _allAirports, _allCountries containers can be any type you wish (i.e. list, set, or dictionary, etc.); However, the _allFlights container MUST be a dictionary.

2. 3 setters: **getAllAirports, getAllFlights, getAllCountries** and 3 getters: **setAllAirports, setAllFlights, setAllCountries** for _allAirports, _allFlights, _allCountries.

3. **loadData(self, airportFile, flightFile, countriesFile):**
   - Read in all the data from the **countries file** of the given name the parameter, countriesFile. Extract the information from each line. Remove any whitespace from the outside of each portion of the line (not just the line itself). To add all the file contents to the _allCountries container, we suggest to create it as a dictionary. For each line in the file you add a new element to the _allCountries dictionary so that the key is the country name while the value is the continent name.
   - Read in all the data from the **airports file** of the given name the the parameter, airportFile. 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 _allAirports 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. Don't forget to use the _allCountries dictionary while creating the Airport objects.
   - Read in all the data from the **flights file** of the given name through the parameter, flightFile. 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 _allFlights **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

TOP

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 all 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 and make sure to close the files properly.

When reading the text files, use the following line

f = open(filename, "r", encoding='utf8')

The encoding parameter should be 'utf8' (nevermind why should we use this).

4. **getAirportByCode(self, 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, just return -1.

5. **findAllCityFlights(self, city)**

   Return a list that contains all Flight objects that involve the given city **either as the origin or the destination**

6. **findFlightByNo(self,flightNo)**

   Returns a flight object of with the flight number equals to flightNo. Return -1 if not found.

7. **findAllCountryFlights(self, country)**

   Return a list that contains all Flight objects that involve the given country **either as the origin or the destination (or both).**

8. **findFlightBetween(self, origAirport, destAirport)**

   - Check if there is a direct flight from origAirport object to destAirport object. If so, return a string of the format:

     ```
     Direct Flight(flightNo): origAirportCode to destAirportCode
     ```

     i.e. Direct Flight(ABC456): YYZ to ORD

   - Otherwise, if there is **no direct flight**, check if there is a single-hop connecting flight from origAirport to destAirport. This means a sequence of exactly (2 flights, 3 airports) such that the first flight begins in origDestination and ends in some airport "X", and the second flight goes from airport "X" to destAirport. like:

     origAirport ----> X ----> destAirport

     Create and return **a set** (not a list) of **all possible "X" airport codes** representing the airports that could serve as the connecting airport from origAirport to destAirport. **Do not include the origAirport and destAirport in the set**. Do not worry about multiple-hop connections as that becomes very complicated to track. You should only look at single-hop connecting flights. Like, we are not interested in such flights:

     origAirport ----> X ----> Y ----> destAirport

     origAirport ----> X ----> Y ----> .... ----> destAirport

     Again, this should only be done if there was no direct flight.

TOP

- If there is no direct flight AND no single-hop connecting flights from origAirport to destAirport, then just return -1.

9 **findReturnFlight(self, firstFlight)**

Take the given Flight object (firstFlight) and look for the Flight object representing the return flight from that given flight. In other words, given firstFlight from origin A to destination B, find the Flight object that departs from origin B and arrives in destination A. The method should return this returning flight object. If there is no such Flight object that goes in the opposite direction as firstFlight, just return -1.

10 **findFlightsAcross(self, ocean)**

This method takes an ocean name as a parameter. The value of this parameter is either of two values: "Atlantic", "Pacific" as strings. You can assume that this value is always sent correctly.

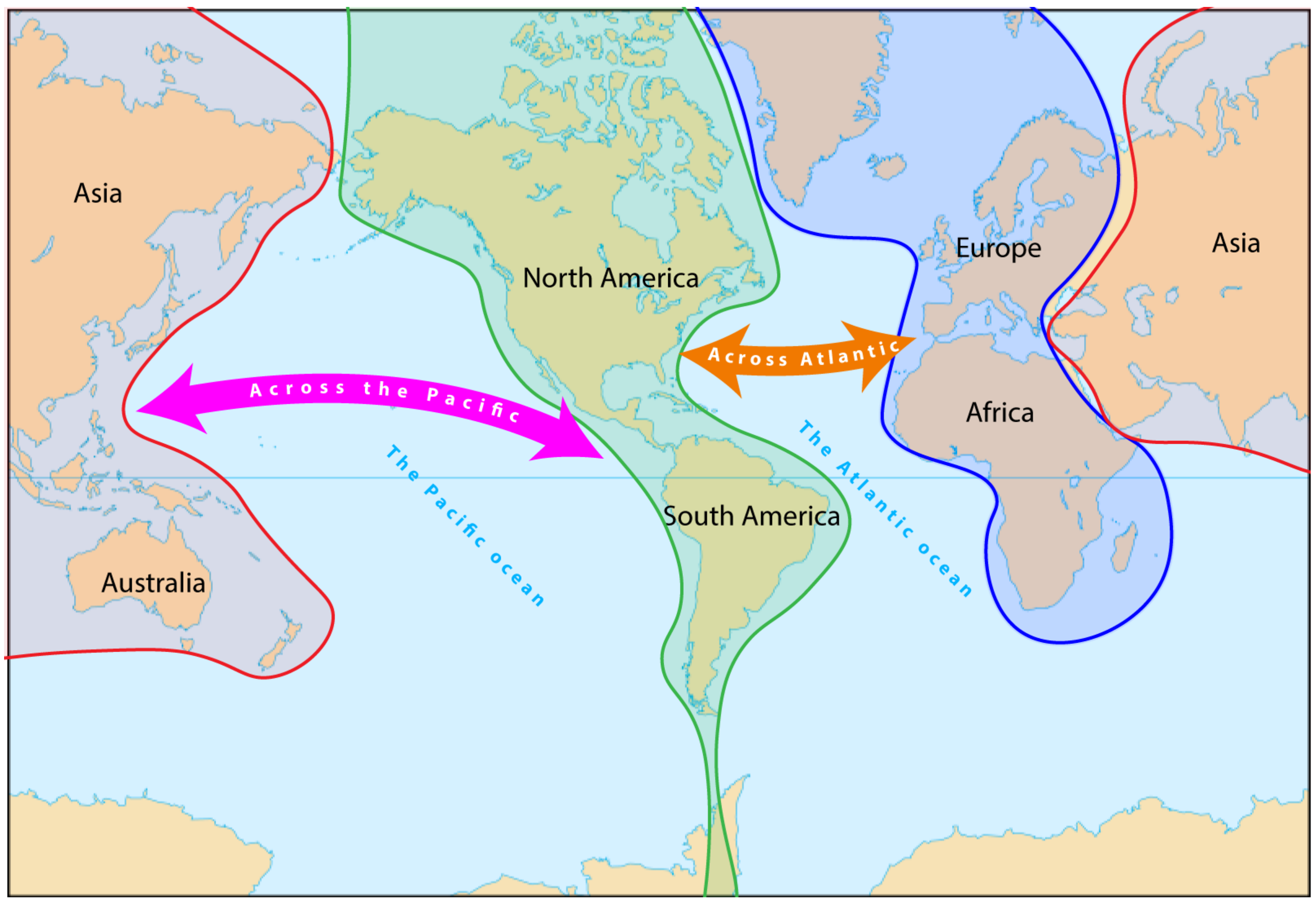
In the image below, you can see the world map divided into three zones.

- North America and South America in the green zone.
- Asia and Australia in the red zone.
- Africa and Europe in the blue zone.

A flight crosses the Pacific if the origin is the from one of the countries in the green zone and the destination is to one country in the red zone or vice versa.

On the other hand, **a flight crosses the Atlantic if the origin is the from one of the countries in the green zone and the destination is to one country in the blue zone or vice versa.**

This method will return a **set** (not list) of **all** the flight codes that cross the specified ocean, or return -1 if there is no flights.



TOP

Tester File (Asst4Tests.py)

You are provided with a file called Asst4Tests.py. This file will run tests to check that each of your classes and methods are working as expected. You can uncomment the tests one by one to test the functionalities separately and isolate potential problems.

GRADESCOPE TESTING!

It is your responsibility to do thorough testing to ensure the code works not just for the given tests and data but for **many different test cases and different sets of data.**

DEBUGGING!

It is very highly encouraged to use **the debugging tools** for this assignment. Remember, this assignment includes a complex structure, so, 80% of the time spent in this assignment is on fixing problems.

Gradescope may also use different files (same format but different names and data) so your program should not have anything hardcoded.

If you are failing one or more tests, you should look at what the test is checking for and each of the sub-tests within that test (most tests are checking 2-4 different cases to ensure it works for all those cases). It may help to print out each of test case values, i.e.

print(t1)

print(t2)

print(t3)

You should also add other print lines to help debug and narrow down the source of the errors in your code. Make sure to remove these temporary print lines once finished debugging.

You **can** also create your own testing environment and test your code as you're developing it. You then can use Asst4Tests.py after you complete the code.

USER INPUT!

The three files that you need to submit does NOT ask for user input at any point. They are basically consist of class definitions and function definitions.

EXCEPTION HANDLING!

Remember to handle all expected exceptions in your code. Failure to do so may result in incomplete execution of your code in Gradescope and potential marks loss.

TOP

Functional Specifications

Your program must work for any airports file and flights file. The format will be the same, but they could have different names and a different list of airports or flights. Your program must work with the files of the given names, so do not hardcode it to only work for the files provided to you.

The Asst4Tests.py file is a great way to test that your functions are working according to specifications. However, you must do your own testing in addition to the tests given in this file. The Gradescope autograder may run additional tests that are not visible to you, so your program should be well-tested to ensure it works not just for the given tests but in a large multitude of test cases.

Non-Functional Specifications

- The program should strictly adhere to the requirements and specifications of these classes and functions.
- The program should include brief comments in your code identifying yourself, describing the program, and describing key portions of the code.
- Assignments are to be done individually and must be your own work. Software may be used to detect academic dishonesty (cheating).
- Use Python coding conventions and good programming techniques. For example: Meaningful variable names
    - Conventions for naming variables and constants
    - Use of constants where appropriate
    - Readability, indentation, and consistency
- The name of the files you submit must be Airport.py, Flight.py, and Aviation.py

2 TIPS AND GUIDELINES

- Variables should be named in lowercase for single words and camel case for multiple words, i.e. origAirport
- Do not hardcode filenames, numbers of lines from the files, or any other variables
- You can assume that the file formats for each of the 2 types of text files will be the same for any file being used for testing. You may not assume that the number of lines nor the values stored in them will be the same.
- Add comments throughout your code to explain what each section of the code is doing and/or how it works

3 RULES

- Read and follow the instructions carefully.
- Only submit the Python file described in the Files section of this document.

TOP

- Submit the assignment on time. Late submissions will receive a late penalty of 10% per day .
- Forgetting to submit a finished assignment is not a valid excuse for submitting late.
- Submissions must be done on Gradescope. They will not 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 subject to penalties.
- 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.**

4 MARKING GUIDELINES

The assignment will be marked as a combination of your auto-graded tests (both visible and hidden tests) and manual grading of your code logic, comments, formatting, style, etc. Below is a breakdown of the marks for this assignment:

- **[50 marks]** Auto-graded Tests
- **[20 marks]** Code logic and completeness
- **[10 marks]** Comments
- **[10 marks]** Code formatting
- **[10 marks]** Meaningful and properly formatted variables
- **Total: 100 marks**

REMEMBER!

The weight of this assignment is **10%** of the course mark.

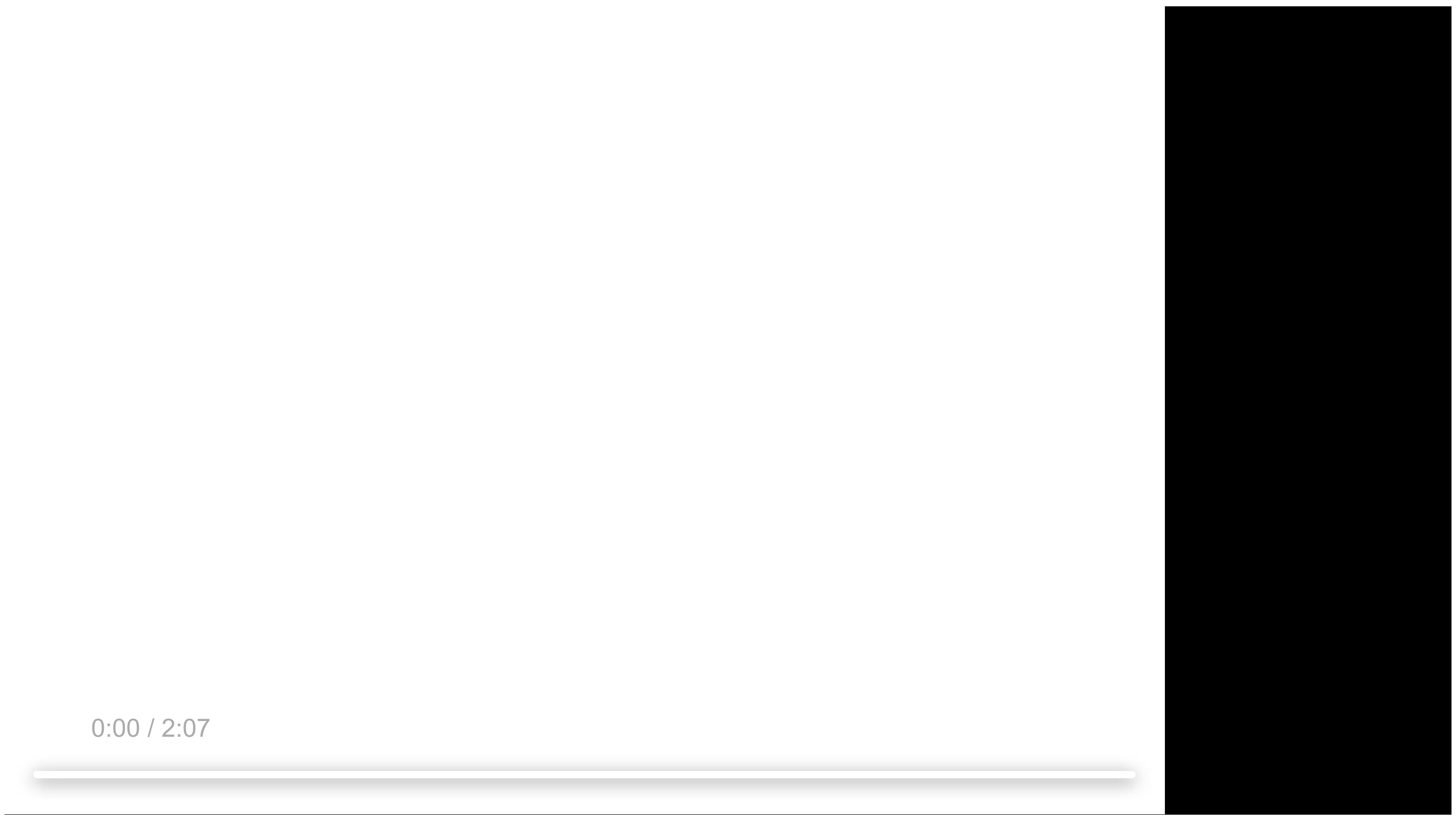
- GRADESCOPE SUBMISSION

1. You must submit the 3 python files to the Assignment 4 submission page on Gradescope. The required files are `Airport.py`, `Flight.py`, and `Aviation.py`. Please, **dont't** submit and other files.
2. Double check the files names and the extensions to be .py. No other file names or extensions will be considered.
3. Make sure again that there is **no user input, no function calls outside the class, and no unhandled exceptions**. Failure to do so may result in your code being stuch while grading.

TOP

4 The submission will be using Gradescope.

You can find the instruction on how to complete your submission by following the steps in this video:



REMEMBER!

- Assignment submission after **11:55 PM** will cause late penalty of 10% per day to be deducted from your mark.
- Submissions through the email will not be accepted at any circumstances.
- Please check back this page whenever an announcement is posted regarding this assignment.

this file was last modified on 2023-03-28 14:57:30 EST

TOP