CS 112 Assignment 13 – Classes & Exceptions

Due: Sunday, December 6th, 11:59pm

Files:

- Create your own file with our convention.
- You should download the tester file from Piazza and run it as always:

```
python3 tester13.py yourfile.py just these classes
```

  - Note: you can only use three `class names` as the targeted tests for this lab:
    - `Plane` testing everything with `Plane` class w/o exception behavior
    - `Hangar` testing everything with `Hangar` class w/o exception behavior, must have `Plane` class implemented
    - `PlaneError` testing `PlaneError` class and the exception behavior of `Plane` class and `Hangar` class

---

Classes allow us to define entirely new types. We might think of each type in Python as a set of values, and we use a class declaration to describe how to build and interact with all the values of this brand new type. We will often need at minimum a constructor (telling us what instance variables to always create), __str__ method (telling Python how to represent the thing as a string), and then any extra methods that we deem useful ways of interacting with the new values.

Turning It In

Add a comment at the top of the file that indicates your name, userID, G#, lab section. Once you are done, run the testing script once more to make sure you didn't break things while adding these comments. If all is well, go ahead and turn in just your one .py file you've been working on over on BlackBoard to the correct lab assignment. We have our own copy of the testing file that we'll use, so please don't turn that in (or any other extra files), as it will just slow us down.

What can I use?

You may NOT import any module.

You may NOT use anything that hasn't been covered in class. List comprehensions, lambda functions, generators, etc. If you aren't sure, ask.

Grading Rubric

```
Pass shared test cases        4x25 (zero points for hard-coding)
   ■ read the instructions carefully as some deductions are possible
----------------------------------
TOTAL:                        100
```

Task 1 – Plane

A `Plane` represents a specific airplane that be parked in a hangar.

`class Plane:` Define the `Book` class.

- `def __init__(self, model, manufacturer, fuel):` `Plane` constructor. All three parameters must be stored to instance variables of the same names (`model`, `manufacturer`, and `fuel`).
  - `model` :: `str`. Represents the model number of the Plane, like `"F-16"` or `"747"`.
  - `manufacturer` :: `str`. Represents the company that made the plane, like `"Boeing"`.
  - `fuel` :: `float`. Represents amount of fuel in the plane as a percentage of its tank size. Must be between 0 (minimum) and 100 (maximum) inclusive. If the provided fuel is outside of this range (for example, if `fuel == "-12.5"`), then raise a `PlaneError` with the message `"bad fuel -12.5"`. (You can skip this part until you get to Task 3).
- `def __str__(self):` returns a human-centric string representation. If `model=="F-16"`, `manufacturer=="General Dynamics"`, and `fuel=="34.77"`, then the returned string must be `"F-16, General Dynamics :: 34.77 / 100"` (note the characters that aren't a part of the data).
- `def __eq__(self,other):` we want to check that two planes are equal (our `self` and this `other` plane). The model and manufacturer of each plane must be the same for the planes to be considered equal. Two planes can have different amounts of fuel and still be considered the same.
- `def is_empty(self):` checks whether this plane has zero fuel. Return `True` if there is zero fuel; return `False` otherwise.
- `def refuel(self, amount):` Attempts to add an amount of fuel to the plane. You cannot refuel by a negative amount. You cannot refuel by an amount that would put the plane above 100% fuel. Should either of these things happen (for example, you try to refuel by -16.8), you should raise a `PlaneError` with the message `"unable to refuel by -16.8"`. (Again, you can skip this part until you get to Task 3).

Task 2 - Hangar

This represents a group of `Plane` values as a list (named `planes`). We can then dig through this list for useful information and calculations by calling the methods we're going to implement.

`class Hangar:` Define the `Hangar` class.

- `def __init__(self, name):` `Hangar` constructor. Create an instance variable to store the name of the Hangar. Create another instance variable, a list named `planes`, and initialize it to an empty list. This means that we can only ever create an empty `Library` and then add items to it later on.
- `def __str__(self):` returns a human-centric string representation. We'll choose a multi-line representation (slightly unusual) that contains `"Hangar Name:"` on the first line, replacing the name of the Hangar accordingly, and then each successive line is a tab, the `str()` representation of the next `Plane` object in `self.planes`, and then a newline each time. This means that the last character of the string is guaranteed to be a newline (regardless of whether we have zero or many `Plane` values).
- `def __eq__(self,other):` Two hangars are equal if they contain the same planes in the same order. This method will check if tow hangars are equal. (our `self` and this `other` hangar). The only things that we need to compare are their lists of planes. Note: you can compare two lists `l1` and `l2` directly as `l1==l2` but this will rely on your implementation of `__eq__()` for `Plane` class since we are comparing two lists of books.
- `def add_plane(self, plane):` append the argument `plane` to the end of `self.planes`. In order to ensure every plane is unique, if our hangar already has a plane of the same model, manufacturer, and fuel raise a `PlaneError` with the message `"duplicate plane`

`'Model:Manufacturer'"`. Be sure to replace Model and Manufacturer with the correct information about the plane that caused the error. (You can skip this exception-raising part until later).

- `def plane_by_model(self, model):` Look through all stored `Plane` objects in `self.planes`. Return the `Plane` object whose title matches the `model` argument. If no such plane exists, raise a `PlaneError` with the message `"no plane found with model 'Model'"`. Be sure to replace Model with the appropriate information about the Plane that caused the error. You can assume that every model in a hangar is unique.
- `def planes_by_manufacturer(self, company):` search through `self.planes` in order, return a new list of `Plane` objects in this `Hangar` that were built by `company`. If no planes were built by this company, raise a `PlaneError` with the message `"no planes built by 'Company'`. Be sure to replace Company with the appropriate information about the company that caused the error.
- `def total_empty(self):` Look through all stored `Plane` objects in `self.planes`. Return the number of all planes that *are out of fuel* ***(hint: a method in Plane helps here.)***
- `def refuel_all(self, amount):` Attempt the refuel all of the planes in the hangar by the amount given. If a plane is unable to be refueled by the amount, you should simply skip over that plane an try to refuel the next one. **You may not write any if statements in this method.** There is another construct (try/except) that you should instead. ***(hint: a method in Plane helps here)***

Task 3 – PlaneError

The `PlaneError` class extends the notion of an `Exception` (note the `(Exception)` part of the class signature line). We're making our own exception that stores a single string message.

`class PlaneError(Exception):` Define the `PlaneError` class to be a child of `Exception` class (by putting `Exception` in the parentheses as shown).

- `def __init__(self,msg):` Constructor. Store `msg` to an instance variable named `msg`.
- `def __str__(self):` the human-centric representation of `PlaneError` is just `self.msg`

If you skipped the exception-raising parts above, you should now go back and add those checks/raises. The test cases for `PlaneError` will go back and check if those parts correctly did so.
