MTH 3300
E. Fink

3300 Problems, Section 7: Object-Oriented Programming

1. Consider a class called `Circle`, whose objects have the following attribute variables:

   - `._rad`, representing the *radius* of the circle;
   - `._h`, representing the *x-coordinate of the center*;
   - `._k`, representing the *y-coordinate of the center*;

   and the following methods:

   - a constructor which takes three outside arguments, which set `_r`, `_h`, and `_k`;
   - `.isInside()` which takes two outside arguments named `xC` and `yC` as, and returns `True` if the point (`xC`, `yC`) is within distance `_r` of the center (`_h`, `_k`), and false otherwise.

   (Recall the distance formula $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.)

   a. Suppose that the class `Circle` works. Complete the following client code so that the phrase `Inside!` is printed to the console if (`myX`, `myY`) is within the circle you declared.

```
my_circ = Circle(3,4,2)
myX = float(input("Enter an x-coordinate: "))
myY = float(input("Enter an y-coordinate: "))
```

   b. Now, write a definition for the class `Circle` that meets the given specifications and that makes the above code work.

   c. Add another method so that the following code causes `"(x - 3)^2 + (y - 2)^2 = 25"` to print to the console:

```
my_circ = Circle(5,3,2)
print(my_circ)
```

2. Consider a class called `Line` whose objects have the following attribute variables:

   - `._m`, representing the slope of a line $y = mx + b$;
   - `._b`, representing the y-intercept of a line $y = mx + b$;

   and the following methods:

   - a constructor which takes two outside arguments, which set `_m` and `_b` respectively;
   - `.yInt()`, a method which has no outside arguments and returns the y-intercept of the `Line`;
   - `.xInt()`, a method which has no outside arguments and returns the x-intercept of the `Line`;
   - `.y_value()`, a method which takes an `x` value as outside input, and returns the y coordinate of the point of the `Line` with that `x` value.

   a. Write the definition for the class `Line`, with methods as described.

   b. Suppose that the class `Line` works. Create a `Line` object called `pretty_line` that represents the line $y = 4x + 2$. Then, print the x- and y-intercepts of `pretty_line`. (Obviously, use the methods you implemented.)

   c. Suppose that `x_vals = [4.1, 5.2, 8.3, 9.1, 6.1]` is a list of x-values. Write a loop that goes through this list, and prints out the corresponding y-values of to these x-values on `pretty_line`. Again, use the method you implemented.

3. Consider a class called `Student` whose objects have the following attributes:

   - `._name`, representing the *name* of the student;
   - `._scores`, representing a `list` of test scores;

   and the following methods:

   - a constructor which takes in a `str` and a list of `float`s, which sets the attribute variables;
   - `.average()` which takes no (outside) arguments, and returns the average of the test scores;

- `.passes()` which takes no (outside) arguments, and returns `True` if the average was greater than or equal to 60 and `False` otherwise.

a. Write the class definition for the class `Student`. Write the function `.passes()` without making any direct reference to `._scores`.

b. In client code, create a new `Student` object named `x`, whose `name` is `Evan`, with scores `20`, `30` and `40`; and then print out this `Student`'s average using the member function.

c. Add one more method, which overloads the `<` operator: this should be implemented so that `s1 < s2` when `s1`'s average is less than `s2`'s average.

4. Consider a class called `Truck`, meant to model an individual truck in a company's shipping fleet. The objects of this class should have the following attribute variables:

   - `._max_items`, representing the **maximum** number of items that the truck can hold;
   - `._max_weight`, representing the **maximum** weight of items that the truck can hold;
   - `._cur_items`, representing the **current** number of items that the truck can hold;
   - `._cur_weight`, representing the **current** total weight of items that the truck can hold;

   and the following methods:

   - a constructor which takes two arguments, the max number of items and the max weight for a given truck, which sets all four attribute variables – *assume that every truck starts out as empty*;
   - `.add_on()`, which takes one `float` named `w` as an (outside) argument. This function should attempt to add 1 item with weight `w` to the truck: specifically, if the item doesn't cause the truck's number of items or weight to exceed the maximums, the function should update the member variables appropriately, and return `true`; otherwise, the function shouldn't update any variables and return `false`.

   a. Write the class definition for the class `Truck`.

   b. Write client code that does the following: create a new `Truck` object named `biggie`, whose weight capacity is `15000` and which can hold `80` items. Then, have the user input a weight for an item. The program should then add on an item with that weight to `biggie` – and if the item is "rejected" because it is too heavy, the program should print out a message that says `FAIL`.

   c. Add one more method, which overloads the `<` operator: this should be implemented so that `t1 < t2` when `t1`'s current weight is less than `t2`'s.

5. WaitList is a program that helps doctor's offices manage their wait-lists.

   Consider a class called `Waitlist`. Each object is meant to represent a wait-list: a list of names of patients, to be seen in order. Each `Waitlist` object should have the following **attribute variables**:

   - `._patients`, a list of strings, representing the **names of patients** (for example, `["Bob B", "Alice A", "Joe J"]`);
   - `._num_patients`, an integer, representing the number of patients in the list.

   The class should also support the following **methods**:

   - a constructor which takes NO (outside) arguments: it should just initialize `._patients` to be an empty list and `._num_patients` to be 0;
   - `.add()`, which takes one string (a name) as outside argument; this function should return nothing, but add the name to the end of `._patients`, and add 1 to `._num_patients`;
   - `.call()`, which takes no (outside) arguments. If there is at least one patient, this function should remove the FIRST patient from the waiting list (that means removing the name from `._patients`, and lowering `._num_patients` by 1), and return the name of that patient. If there are no patients, the function should return the string `""` and do nothing else.

   a. Write the class definition for the class `Waitlist`.

   b. Create a `Waitlist` object named `www`. Then, add two patients to it: one named `Evan`, and one named `Frank`.

   c. Suppose that `www` has accumulated more patients. Write a loop which prints out all the patients in the `Waitlist`, in order, until there are no mor patients. **You should do this WITHOUT referencing the attribute variables** `._patients` and `._num_patients`; instead, use the method `.call()`.

d. Add one more method, so that if I were to `print(www)` where `www` was as in part b, it would cause

```
Evan
Frank
```

to appear on the screen.

6. Consider a class called `Section`, meant to model an individual section of a college course. The objects of the class should have the following attribute variables:

   - `._capacity`, representing the **maximum** number of participants that can be in the section (including the instructor);
   - `._current_num`, representing the **current** number of participants (including the instructor);
   - a list of `str`s named `._participants`, representing the **current** list of the names of the participants (including the instructor);

   and the following public members:

   - a constructor which takes two arguments, the section capacity and the name of the instructor, which initializes the three attribute variables – *assume that the section starts out as containing only the instructor as participant*;
   - a function `.enroll()`, which takes one `string` named `s` as an (outside) argument. This function should attempt to enroll a student with name `s` to the section: specifically, if the extra student doesn't cause the section's current number of participants to exceed the capacity, the function should update the member variables appropriately, and return `True`; otherwise, the function shouldn't update any variables and return `False`.

   a. Write the class definition for the class `Section`.

   b. In client code, suppose that an `int` variable `cap` and a list of names called `new_students` *have already been initialized*. Write code that does the following: create a new `Section` object named `ef3300`, whose capacity is `cap` and whose instructor is `Evan`. The program should then attempt to add all the students from `new_students` to `ef3300` – and if the any of the students is "rejected" because the class is full, the program should print out a message that says `NOT ENROLLED`.

   c. The function `__iadd__` overloads the `+=` operator in Python. Implement this operator so that if `s` is a `Section` and `n` is an integer, then `s += n` increases the capacity of `s`. (Warning – overloading `+=` in Python is a little counterintuitive, given the examples I've shown you – you might want to look this up.)