APPY Exercises on OOP
Part I

The questions are based on the exercises in the book **Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and The Cloud by Paul Deitel and Harvey Deitel. Pearson, 2020** and the book of additional material.

Questions

1. Modify the class `Point` as follows (check the posted `exercises.py` file):
   (a) Declare `x` and `y` as private properties, and define their setters and getters using decorators.
   (b) Include `__repr__` and `__str__` methods that return the string representations of a `Point` instance.
   (c) Add method `distance_to` that takes another point as an argument and calculates the distance between the two points.
   (d) The equation of a straight line is (`y = m x + c`). The coefficients `m` and `c` completely describe the line. Write a method `calculate_equation` that takes another point as argument. The method computes the equation of the straight line between the two points and returns the two coefficients as a tuple of the form `(m, c)` (refer to this link to know how to calculate these coefficients).
   (e) Add method `move_to` that receives x- and y-coordinate values and sets the `Point`'s new location to the given coordinates.

2. A circle is completely described by at its center point and its radius. In that view, implement the following:
   (a) Create a class `Circle` that has as its attributes `radius` and `point` (a `Point` that represents the `Circle`'s center location). Declare those variables as private properties and define their setters and getters using decorators.
   (b) Include `__init__`, `__repr__`, and `__str__` methods for `Circle` class.
   (c) Add a read-only property `area`. Do not add an attribute for it, rather compute it from the attributes of the class `Circle`.
   (d) Add method `move_to` that receives x- and y-coordinate values and sets a new location for the `Circle` by calling the composed `Point` object's `move_to` method.
   (e) Add a method `is_touching` that takes another `Circle` as an argument. The method checks whether the two circles are touching/intersecting, or whether they are separated apart. (In other words, the method returns `True` when the distance between the centers of the two circles is less than or equal the summation of their radii. Otherwise, the method returns `False`).

3. Create a new class named `SMSsStore`. The class instantiates `SMSsStore` objects that are similar to an inbox or outbox on a cellphone. This store can hold multiple SMS messages (i.e. its internal state will just be a list of messages). This is represented by the attribute `messages`. Each message will be represented as a tuple of the form

   ```
   (has_been_viewed, from_number, time_arrived, text_of_msg)
   ```

   where `has_been_viewed` is a boolean, `from_number` is a string of digits, `time_arrived` is a python `datetime` object (refer to this link), and `text_of_msg` is a string representing the text of the message.

   Create the class and augment you class with the following:

   (a) Include method `__init__` to construct a new instance of an `SMSsStore`.

   (b) Include method `__repr__` to override the default string representation of an `SMSsStore` instance.

   (c) Add method with the following signature `add_new_arrival(self, from_number, time_arrived, text_of_SMS)` that when invoked on a an instance of `SMSsStore`, it makes new SMS tuple, inserts it after other messages in the store. Note that when creating this message, its `has_been_viewed` status is set `False`. The data types of arguments is as mentioned before.

   (d) Add method `message_count` that returns the number of SMS messages in the invoked `SMSsStore` object.

   (e) Add method `get_unread_indexes` that returns a list of indexes of all not yet viewed SMS messages.

   (f) Add method `get_message` that takes an integer `i` as an argument. The method returns `(from_number, time_arrived, text_of_SMS)` for the `i`th message. It also changes its state to `has_been_viewed`. If there is no message at position `i`, then the method returns `None`.

   (g) Add method `delete` that takes an integer `i` as an argument. The method deletes the message at index `i`. If there are no messages at the given index, then you have to raise a `ValueError` with the following custom message

   `"You requested to delete the message at index i, while there are only n messages"`

   , where `i` should be replaced by the given index, while `n` should be replaced by the number of messages in the store.

   (h) Add method `get_messages_from` that takes a string `number` as an argument. The method returns a list of `text_of_SMS` for the messages with `from_number` equals to `number`. It also changes all their states to `has_been_viewed`.

   (i) Add method `clear` that deletes all messages of the invoked `SMSsStore` object.

   Note: Test cases for this class are not provided so that you can try to write your own.