Worksheet 2 - Solution

Miscellaneous Exercises

Problem 1:
For the following code

```
class Car:
  def __init__(self, color, model, year):
    self._color = color
    self.__model = model
    self.__year__ = year

c = Car("red", "Toyota", 2020)
```

Which of the following statements will result in an error:

Statement	**Valid? (Yes/No)**	**Explanation**
`print(c._color)`	Yes	prints the `_color` attribute, It is not name mangled.
`print(c.__model)`	No	The __model attribute is name mangled, you can only access it through `_Car__model`
`print(c.__year__ )`	Yes	trailing underscores prevents name mangling even if the name starts with a double underscore.
`c._Car_color = "green"`	Yes	Python will create a new attribute. This is completely separate from the `_color` attribute. `_color` is still “red”
`c._Car__model = "BMW"`	Yes	This will modify the `__model` attribute that is defined in the

Continue to the next page...

		class.
`c._Car__year__ = 2019`	**Yes**	Python will create a new attribute. This is completely separate from the `__year__` attribute. `__year__` is still 2020
`c.__model = "BMW" #THIS ONE IS TRICKY`	**Yes**	This will actually create a new attribute called `__model`. This is going to be separate from `_Car__model` that is name mangled and defined inside the class. Use dir() to check that yourself!

Capstone Project: MaxHandWins

Problem 2:
It is a good practice to make all your attributes private and access them through getter methods. A getter method is a very simple method that returns the value of a specific attribute, and it usually starts with the `get_` prefix.

Go back to all the classes that you defined for the MaxHandWins game and make all your attributes private. Allow access to these attributes through getter methods.

Solution:

Here is how the PlayingCard class will look like.

```
class PlayingCard:
    def __init__(self, rank, suit):
        self._rank = rank
        self._suit = suit

    def get_rank(self):
        return self._rank

    def get_suit(self):
        return self._suit
```

For the Player class, we will get this:

```
class Player:
    def __init__(self, name):
        self._name = name
        self._hand = []

    def get_name(self):
        return self._name

    def get_hand(self):
        return self._hand
```

For the Deck class, we get this:

```
class Deck:
    def __init__(self):
        self._cards = []
        for i in range(13):
            self._cards.append(PlayingCard(i+2, Suit.SPADES))
            self._cards.append(PlayingCard(i+2, Suit.DIAMONDS))
            self._cards.append(PlayingCard(i+2, Suit.HEARTS))
            self._cards.append(PlayingCard(i+2, Suit.CLUBS))

    def get_cards(self):
        return self._cards
```

Problem 3:
Let's add a setter method `set_hand` to the `Player` class (a **setter method** is a method that sets an attribute, whereas a **getter method** is one that reads an attribute) that sets the value of the `hand` attribute. This method should just get the hand as an argument and sets it on the instance.

Solution:

```
class Player:
    def set_hand(self, hand):
        self._hand = hand
```

Problem 4:
We need to be able to quickly identify the largest card that a player has in his hand. This will help identify who wins the round.

In the `Player` class, create a method `strongest_hand` that returns the strongest `PlayingCard` that this player has in his `hand`. Remember that a player only has two cards in his hand at one time.

Solution:

Since we only have two cards in a single hand, this should be easy...or is it?

Let's revise the algorithm first.

First, if one card has a higher rank then the other, then the card with the higher rank is the stronger card.
However, if the two cards have the same rank, then the card with the stronger suit will be the overall stronger card. Remember that suit strength from the strongest to the weakest goes like this: SPADES, HEART, DIAMONDS, CLUBS

Now since we know we might need to compare suits, it makes sense to be a little smarter about the values that we assign to the SUIT enum. Let's give the highest value to SPADES, followed by HEART, DIAMONDS, and finally CLUBS.

```
class Suit(Enum):
    SPADES = 4
    HEARTS = 3
    DIAMONDS = 2
    CLUBS = 1
```

With that, we can implement the strongest_card method like this:

```
def strongest_card(self):
        if len(self._hand) == 2:
            if self._hand[0].get_rank() > self._hand[1].get_rank():
                return self._hand[0]
            elif self._hand[0].get_rank() < self._hand[1].get_rank():
                return self._hand[1]
            # the two ranks are the same.
            elif self._hand[0].get_suit().value >
self._hand[1].get_suit().value:
                return self._hand[0]
            else:
                return self._hand[1]
```

As you will learn later in the course, the above logic can be immensely simplified. So stay tuned ;)

Problem 5:
For the Deck object, *we need to be able to shuffle the cards in the deck*. Create a method `shuffle` that shuffles the cards in the deck. There are many ways to implement this so any shuffling method you can think of can work.

> 👉**PRO TIP** 👈
> You will find `random.randint` exceptionally handy.

Solution:

There are many ways to shuffle the cards, so don't feel obligated to use this solution. If you have a different solution, try it, and then inspect the cards in the deck and see how "random" they are.

One solution to this problem is to swap the positions of two random cards and repeat this operation for a reasonable number of times.

```
class Deck:
  def shuffle(self):
      for _ in range(200):
        i, j = random.randint(0, 51), random.randint(0, 51)
        self._cards[i], self._cards[j] = self._cards[j], self._cards[i]
```

Problem 6:
Another action that needs to be performed on the `Deck` instance is to draw cards from the deck's `card` attribute. Create a method `draw` that takes an integer argument `n` where n represents the number of cards to be drawn from the deck.
This method should return a list of the `PlayingCards` that were drawn. Assume that we draw from the tail of the list (last element first).

> Example:
> d = Deck()
> d.cards = [1♠, 5♡, 3♢, 10♣, J♢]
> d.draw(2) # returns [10♣, J♢]

Hint1: Return None if the count of the remaining cards in the deck is less than n.

Solution:

```
class Deck:
  def draw(self, n):
        if n > len(self._cards):
            return None

        drawn = []
        for _ in range(n):
            c = self._cards.pop()
            drawn.append(c)
        return drawn
```

This is easy!

We first check if the remaining cards are less than the number of cards that need to be drawn, if true, we return None.

Otherwise, we pop the n cards from the tail of the list and return those.