Lab 17: Advanced OOP

Apr 3, 2018

Main Event

1. Players

This exercise will have you develop a family of players with varying attributes.

1. Create a class called Player:
    - Constructor

      ```
      def __init__(self, name)
      ```

      The class should have two attributes: `name` and `power`. Name should be based on the parameter, while power can be set to an arbitrary integer greater-than zero.

    - As a string

      ```
      def __str__(self)
      ```

      Return the players name and power as a single string.
2. Create three child players:
    1. `FastPlayer`
    2. `TallPlayer`
    3. `CoolPlayer`

    In addition to `name` and `power` each child should have its own unique attribute:

    - FastPlayer should have `speed`
    - TallPlayer should have `height`
    - CoolPlayer should have `awesomeness`

Values for these additional attributes should be passed to the respective constructors.

Further, each child should contain its own `__str__` method that "specializes" what their parent does. That is, each string is a combination of its parents string and its respective attribute.

To test the code, create instances of the four Player types and print each. For example:

```
>>> p = Player('Player 1')
>>> print(p)
Player 1 100
>>> p = FastPlayer('Player 2', 45)
>>> print(p)
Player 2 100 45
```

Solution:

```
import random

class Player:
    def __init__(self, name):
        self.name = name
        self.power = random.randrange(100)

    def __str__(self):
        return self.name + ' ' + str(self.power)

class FastPlayer(Player):
    def __init__(self, name, speed):
        super().__init__(name)
        self.speed = speed

    def __str__(self):
        return super().__str__() + ' ' + str(self.speed)

class TallPlayer(Player):
    def __init__(self, name, height):
        super().__init__(name)
        self.height = height

    def __str__(self):
```

```
        return super().__str__() + ' ' + str(self.height)

class AwesomePlayer(Player):
    def __init__(self, name, awesomeness):
        super().__init__(name)
        self.awesomeness = awesomeness

    def __str__(self):
        return super().__str__() + ' ' + str(self.awesomeness)

p1 = FastPlayer('Usain Bolt', 9.58)
p2 = TallPlayer('Robert Wadlow', 2.72)
p3 = AwesomePlayer('Bart Simpson', 10)
print(p1, p2, p3, sep='\n')
```

2. Playing cards

Card

Implement a class called Card that has two attributes: number and suit. It should also define what it means for one card to be greater-than another by implementing `__gt__`. What this means specifically is up to you, but it should be some function over the attributes. For example, if the values are a tie, then order should be distinguished based on the suit.

```
>>> a = Card(w, x) # where w and x are the number and suit
>>> b = Card(y, z)
>>> a > b
```

The data type your choose to represent the value and suit is up to you.

Deck

Implement a class called Deck that inherits from `list` (the Python internal list type). When constructed, it should add 52 cards of various types to *itself*:

```
card = Card(...)
self.append(card)
```

The class should also have a method `deal` that removes and returns a single card from the deck.

Player

Implement a class called Player. Player should have three attributes: `name`, `card`, and `wins`. Name should be initialized from a parameter, card can be initialized to any value you want (it will get populated later), and wins should be zero.

Player should have two methods:

1. `receive` takes a variable of type `Card` and assigns to the corresponding attribute.
2. `__gt__` determines whether the number of wins by this player is more than the number of wins by the other.

At the table!

As a simple first game, repeatedly deal a single card to two players. Whoever has the better card should have their `win` total incremented. Once all the cards have been dealt, report the player who had the most wins.

Solution:

```
import random

class Card:
    def __init__(self, face, suit):
        self.face = face
        self.suit = suit

    def __gt__(self, other):
        if self.face == other.face:
            return self.suit > other.suit
        return self.face > other.face

class Deck(list):
    def __init__(self):
        for face in range(13):
            for suit in range(4):
                c = Card(face, suit)
                self.append(c)
        random.shuffle(self)
```

```
    def deal(self):
        return self.pop()

class Player:
    def __init__(self, name):
        self.name = name
        self.hand = None
        self.wins = 0

    def receive(self, card):
        self.hand = card

    def __gt__(self, other):
        return self.wins > other.wins

p1 = Player('Player 1')
p2 = Player('Player 2')

d = Deck()
while d:
    p1.receive(d.deal())
    p2.receive(d.deal())

    if p1.hand > p2.hand:
        p1.wins += 1
    else: # assume no two cards are the same
        p2.wins += 1

if p1 > p2:
    winner = p1
else: # assume an odd number of rounds
    winner = p2

print(winner.name, 'wins')
```

Additional Practice

1. Card game

Update your code from the previous exercise such that a player can hold multiple cards. The class will likely require a change to the data type of `hand`, the semantics of `receive`, and logic behind `__gt__`.

In theory, to the user of your class nothing has to change—this is one of the advantages of object oriented programming. However, because this is practice, you are encouraged to change the way your class is used so that a multi-card game is built. That is, rather than dealing a single card, deal several. Note that how this is done, along with how `__gt__` is implemented, will dictate the game that you make.

Solution:

```
# ... Card and Player are the same...

class Player:
    def __init__(self, name):
        self.name = name
        self.hand = []
        self.wins = 0

    def receive(self, card):
        self.hand.append(card)

    def __gt__(self, other):
        self.hand.sort()
        other.hand.sort()

        better = 0
        for (i, j) in zip(self.hand, other.hand):
            outcome = 1 if i > j else -1
            better += outcome

        return better > 0

# ... in the global scope, only dealing is a little different:

d = Deck()
while d:
    for i in range(2): # assume an even number of cards
        for j in (p1, p2):
            j.receive(d.deal())

    # ... everything else remains the same
```

Introduction to Computer Science

Introduction to Computer Science
jerome.white@nyu.edu

jerome-white

Learning computer science concepts through practice.