Solutions to the retake exam in Programming in Python (1TD327) - 2022-06-10, 5 ECTS

2022-06-10

Grading criteria

Part A consisted of 10 multiple choice questions. Each correct answer was worth 1 point, giving 10 points in total. Part B consisted of 15 tasks, one of which was a bonus task. Value of each correct answer is listed next to each task in the exam. In total, there were 32 points available and the bonus task was worth 4 points.

- Grade 3: 8 or more points on Part A
- Grade 4: 8 or more points on Part A and between 19 and 32 points on Part B
- Grade 5: 9 or more points on Part A and 32.5 points or more on Part B

Solutions to Part A

Question	Answer
1	A
2	B
3	C
4	B
5	C
6	B
7	C
8	D
9	B
10	B

Grading system: 1 point per correct answer, 0 if the answer is wrong, missing, or several options were circled.

Solutions to Part B

Simulation of a Simple War card game

***Note**: The solutions presented here are only suggestions, there are other implementations that are acceptable.*

If you have any questions about the solutions presented or the grading, please contact the teachers.

Listing 1: Initialization

```
import random

suits = ['Hearts','Clubs','Spades','Diamonds']

points = {'Two': 2, 'Three': 3, 'Four': 4, 'Five': 5, 'Six' :
    6,
          'Seven' :7, 'Eight': 8,'Nine':9, 'Ten':10,
          'Jack':11, 'Queen': 12,'King': 13,'Ace':14}
```

Listing 2: Card class

```
class Card():
    """
    Models a playing card.
    """
    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank
        self.pts = points[rank]

    def wins(self, adversary_card):
        if self.pts > adversary_card.pts:
          return 1
        elif self.pts == adversary_card.pts:
          return 0
        else:
          return -1

    def __str__(self):
        return self.rank + ' of ' + self.suit
```

Listing 3: Card instance example

```
mycard = Card('Clubs', 'Ace')
print(f'My card is {mycard} and is worth {mycard.pts} points.
    ')

# Output:
My card is Ace of Clubs and is worth 14 points.
```

Listing 4: Deck class

```
class Deck():
  """
  Models a deck of playing cards.
  """
  def __init__(self):
    self.set_of_cards = []
```

```
  def nb_of_cards(self):
    return len(self.set_of_cards)

  def add_cards(self, new_cards):
    for acard in new_cards:
      self.set_of_cards.insert(0, acard)

  def shuffle_cards(self):
    random.shuffle(self.set_of_cards)

  def pick_top_card(self):
    if len(self.set_of_cards) > 0:
      card_to_pick = self.set_of_cards.pop()
    else:
      card_to_pick = None

    return card_to_pick

  def empty_deck(self):
    self.set_of_cards = []

  def __str__(self):
    return '(bottom) ' + ', '.join([card.__str__() for card
   in self.set_of_cards]) + ' (top)'
```

Listing 5: Deck instance example

```
a_deck = Deck()
a_deck.add_cards([mycard, Card('Spades', 'King'), Card('Clubs
    ', 'Queen'), Card('Hearts', 'Three')])

print(f'There are {a_deck.nb_of_cards()} cards in the deck.')
print('\nThe cards are: ', a_deck)

print('\nThe top card is: ', a_deck.set_of_cards[-1])

print('\nShuffling the cards!')
a_deck.shuffle_cards()
print('\nPicking the top card: ', a_deck.pick_top_card())

print(f'\nNow, there are {a_deck.nb_of_cards()} cards left in
    the deck.')
```

```
# Output:
There are 4 cards in the deck.

The cards are:  (bottom) Three of Hearts, Queen of Clubs,
   King of Spades, Six of Clubs (top)

The top card is:  Six of Clubs

Shuffling the cards!

Picking the top card:  Three of Hearts

Now, there are 3 cards left in the deck.
```

Listing 6: SimpleWar class

```
class SimpleWar():
  """
  Methods for simulating a simple war card game.
  """

  def __init__(self, name1, name2):
    # Create players
    self.player1 = {'name': name1, 'deck': Deck()}
    self.player2 = {'name': name2, 'deck': Deck()}

    # Create main deck of cards with all 52 cards
    self.main_deck = Deck()
    for s in suits:
      for p in list(points.keys())[:3]:
        new_card = Card(s, p)
        self.main_deck.add_cards([new_card])
    self.main_deck.shuffle_cards()
    print(self.main_deck)


  def deal_cards(self):
    half = self.main_deck.nb_of_cards() // 2
    cards1 = self.main_deck.set_of_cards[:half]
    cards2 = self.main_deck.set_of_cards[half:]
    self.main_deck.set_of_cards = []
```

```
    self.player1['deck'].add_cards(cards1)
    self.player2['deck'].add_cards(cards2)

  def play_turn(self):
    card1 = self.player1['deck'].pick_top_card()
    card2 = self.player2['deck'].pick_top_card()
    self.main_deck.add_cards([card1, card2])
    print()
    print('Card 1: ', card1)
    print('Card 2: ', card2)
    print(self.main_deck)

    if card1 is None or card2 is None:
      print('\nGame over!')
    else:
      if card1.wins(card2) == 1:
        self.player1['deck'].add_cards(self.main_deck.
   set_of_cards)
        self.main_deck.empty_deck()
        print(f"{self.player1['name']} wins the turn")
      elif card1.wins(card2) == -1:
        self.player2['deck'].add_cards(self.main_deck.
   set_of_cards)
        self.main_deck.empty_deck()
        print(f"{self.player2['name']} wins the turn")
      else:
        print('War!')

    return self.player1['deck'].nb_of_cards(), self.player2['
   deck'].nb_of_cards()

  def winner(self):
    if self.player1['deck'].nb_of_cards() > self.player2['
   deck'].nb_of_cards():
      return self.player1
    else:
      return self.player2
```

Listing 7: SimpleWar game example

```
game = SimpleWar('Piff', 'Puff')

print(game.player1)
```

```
print(game.player2)

game.deal_cards()
print(game.player1['deck'])
print(game.player2['deck'])

n1, n2 = game.player1['deck'].nb_of_cards(), game.player2['
    deck'].nb_of_cards()

max_n_turns = 100
turn_n = 0

while (n1 != 0 and n2 != 0) and turn_n < max_n_turns:
  n1, n2 = game.play_turn()
  turn_n += 1

print(f"\nAnd the winner is... {game.winner()['name']}!")

# Output
{'name': 'Piff', 'deck': <__main__.Deck object at 0
    x7f6c11642e50>}
{'name': 'Puff', 'deck': <__main__.Deck object at 0
    x7f6c11642850>}

(bottom) Two of Clubs, Two of Diamonds, Four of Diamonds,
    Four of Clubs, Two of Spades, Two of Hearts (top)
(bottom) Three of Diamonds, Four of Hearts, Three of Clubs,
    Four of Spades, Three of Spades, Three of Hearts (top)

...
Card 1:  Three of Hearts
Card 2:  Three of Diamonds
(bottom) Three of Diamonds, Three of Hearts (top)
War!

Card 1:  Three of Clubs
Card 2:  Two of Clubs
(bottom) Two of Clubs, Three of Clubs, Three of Diamonds,
    Three of Hearts (top)
Piff wins the turn

Card 1:  Three of Spades
Card 2:  Four of Hearts
(bottom) Four of Hearts, Three of Spades (top)
Puff wins the turn
```

```
Card 1:  Four of Spades
Card 2:  Two of Hearts
(bottom) Two of Hearts, Four of Spades (top)
Piff wins the turn

Card 1:  Two of Spades
Card 2:  Four of Hearts
(bottom) Four of Hearts, Two of Spades (top)
Puff wins the turn
...

And the winner is ... Piff!
```

Plagiarism detection at an art school

B.15. [5 points]

Several solutions were possible for the task. A solution that scored all possible points needed to have the following features:

- Correctly handles the list of lists input.
- Correctly handles boundaries.
- Correctly counts the cheaters.
- No major misuse of Python data structures and functions.

Listing 8: Conversion between liters and medieval measurement system.

```
#Implementation with Python lists.
def detect_cheaters(table, t):
    nrows=len(table)
    ncolumns=len(table[0])
    neighs=[-1, 1] #Neighbours to check

    #Record which students are cheaters
    cheaters=[[False for _ in range(0, ncolumns)] for _ in
   range(0, nrows)]

    #Go over rows and columns
    for i in range(0, nrows):
        for j in range(0, ncolumns):
```

```
            for rn in neighs:
                ic=i
                jc=j+rn

                if(0<=jc and jc<ncolumns):
                    if(abs(table[i][j] - table[i][jc])<t):
                        #Mark both students as cheaters
                        cheaters[i][j]=True
                        cheaters[i][jc]=True
                ic=i+rn
                jc=j
                if(0<=ic and ic<nrows):
                    if(abs(table[i][j] - table[ic][j])<t):
                        #Mark both students as cheaters
                        cheaters[i][j]=True
                        cheaters[ic][j]=True

    #Now count the number of cheaters
    count=0
    for i in range(0, nrows):
        for j in range(0, ncolumns):
            if(cheaters[i][j]):
                count=count+1

    return count
```

Text wrapping

B.16. [5 points]

Several solutions were possible for the task. A solution that scored all possible points needed to have the following features:

- It correctly handles the cases when the sentence cannot be printed.
- It correctly handles the extra space character at the end of a line.
- It correctly determines the number of lines needed.
- No major misuse of Python data structures and functions.

Listing 9: One pass solution for alliteration problem.

```
def wrap_text(text, W):
```

```
    n_lines=[-1]*W
    words=text.split()

    #Precalculate the word lengths to avoid recalculating
   over and over again in the for loop
    w_lens=[len(word) for word in words]
    N_words=len(words)
    max_len=max(w_lens)

    #Run the calculation from the smallest viable width on
    for w in range(max_len, W+1):
        pos=0
        n_lines_w=0
        curr_w=0
        while pos < N_words:
            #We can only add the word to the current line if
   the line will still fit
            #in the display window after we add the word.
            if(curr_w+w_lens[pos]+1 <= w+1):
                #The word fits.
                curr_w=curr_w+w_lens[pos]+1
                pos=pos+1
            else:
                #The word does not fit, add a new line and
   attempt to add the word again
                n_lines_w=n_lines_w+1
                curr_w=0

        #Count the final line we have been working on when we
    run out of words
        n_lines[w-1]=n_lines_w+1

    return n_lines
```