Solutions to the exam in Programming in Python (1TD327) - 2022-01-07, 5 ECTS

2022-01-07

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 16 and 27 points on Part B
- Grade 5: 9 or more points on Part A and 27.5 points or more on Part B

Solutions to Part A

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

Qwixx game

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

Grading criteria for the solutions to the tasks

B.1. [1 point]

Definition of the attribute `sides`: use an input parameter *sides* or fix the attribute to 6 in the constructor. `self` represents a die object in the scope of the class. The constructor does not have any return value.

B.2. [2 points]

The `__str__(self)` method must return a string. Output example in the code listing 2.

B.3. [2 points]

The `roll(self)` method does not return any value. Output example in the code listing 2.

B.4. [1 point]

The attribute `numbers` is a list of numbers (not strings) that are sorted depending on the value `sort_desc` attribute as shown on Figure 1. the function `range(start, stop)` must be used correctly i.e. it generates integers in $[start, stop)$ (the last number is $stop - 1$) with an increment of 1 if nothing else is specified. Preferably avoid using a `for`-construction for sorting the numbers but rather use list-dedicated functions as `sorted(...)` or `reverse(...)`. Instantiating examples are given in the code listing 4. `self` represents a color row object in the scope of the class. The constructor does not have any return value.

B.5. [2 points]

The `markable(self)` method handles three cases, preferably by using the syntax `if-elif-else`: if numbers have already been marked off and the color row has descending order (12 to 2), only numbers smaller than the smallest one marked off can be played. Else if numbers have already been marked off and the color row has ascending order (2 to 12), only numbers larger than the largest one marked off can be played. Otherwise, the list of numbers that can be played is the one defined in the constructor.

B.6. [2 points]

The `mark(self, num)` method does not return anything. It does not replace the marked number *num* with a `'X'` in the `numbers` attribute since the method `__str__()` handles this. It is not necessary to check that the number is not already in the `offmarks` list since the `markable(self)` method does not allow for double-offmarking to happen. Checking is however a sensible precaution for avoiding an incorrect calculation of the points and the score.

B.7. [2 points]

A syntax with a counter incremented in a for-loop is also possible, but preferably use the function `sum(...)`.

B.8. [1 point]

The syntax for defining the attribute `rows` as a dictionary (not a list!) has to be correct i.e. with pairs of keys and values). The numbers in the rows must be sorted as on Figure 1 for each color. `self` represents a player object in the scope of the class.

B.9. [2 points]

The `rolls(self, dicedict)` method must use correctly the dictionary-specific methods `keys()`, `values()`, or `items()`, as well as it must use the correct syntax for accessing keys and values in a dictionary. Output example in the code listing 10: the keys of the dictionary *dicedict* are colors (string format) and the values are instances of the `Dice()` class. This method preferably uses the `roll()` method defined for a die object and it does not return anything.

B.10. [2 points]

Beyond using the `if-else` for returning `True/False` and the for-construction correctly, the method `plays(self, val, hue)` must mark off the number *val* in the row having the color *hue* as shown in the output example in the code listing 10.

B.11. [2 points]

The `tot_points(self)` method must iterates over all color rows of the player with a correct syntax for the usage of a dictionary, and preferably use the `points()` method previously defined for a color row. Output example in the code listing 10.

B.12. [3 points]

The syntax must be correct for defining a dictionary, defining a list, instantiating a player object, and a die object. See code listing 8.

B.13. Bonus task! [4 points]

Use preferably a while-loop, the syntax has to be correct for it. That is, the stopping condition is defined in a general way (avoid assuming there are only three players for example) with a variable initialized before entering the loop AND this variable is updated in the scope of the while-loop for avoiding an infinite execution. See code listings 9 and 7.

Listing 1: Dice class

```
import random

class Dice():
  """ Represent a colored 6-face die """
  def __init__(self, color='white'):
    self.sides = 6
    self.color = color
    self.face = random.randint(1, self.sides)

  def __str__(self):
    return f'color: {self.color}, value: {self.face}'

  def roll(self):
    self.face = random.randint(1, self.sides)
```

Listing 2: Dice instance example

```
green_die = Dice(color='green')
print('Die: ', green_die)
print('Rolling the die...')
green_die.roll()
print('Die: ', green_die)

# Output:
Die:  color: green, value: 2
Rolling the die...
Die:  color: green, value: 6
```

Listing 3: ColorRow class

```
class ColorRow():
  """ Represents a colored row with numbers in (2, 12) """
  def __init__(self, color, sort_desc=False):
    self.color = color
    self.sort_desc = sort_desc
    self.numbers = sorted(range(2, 12 + 1), reverse=self.
   sort_desc)  # replace with 0 the marked off values and put
    them into offmarks
    self.offmarks = []

  def markable(self):
    if self.sort_desc and self.offmarks != []:  # 12 comes
   first, only smallest values are playable
      most_right = min(self.offmarks) - 1
      num_set = [n for n in range(2, most_right + 1)]

    elif not self.sort_desc and self.offmarks != []:  # 2
   comes first, only largest values are playable
      most_right = max(self.offmarks) + 1
      num_set = [n for n in range(most_right, 12 + 1)]

    else:
      num_set = self.numbers

    return sorted(num_set, reverse=self.sort_desc)

  def mark(self, num):
    self.offmarks.append(num)

  def points(self):
    return sum(self.offmarks)

  def __str__(self):
    line = [str(i) if i not in self.offmarks else 'X' for i
   in self.numbers]
    return f'{self.color} row: \t{line} \t Score = {self.
   points()}'
```

Listing 4: ColorRow instance example

```
red_row = ColorRow('red', sort_desc=False)
print(red_row)
```

```
print('Offmarks: ', red_row.offmarks)
print('Free numbers: ', red_row.markable())
print()
red_row.mark(5)
print(red_row)
print('Offmarks: ', red_row.offmarks)
print('Free numbers: ', red_row.markable())

# Output:
red row:  ['2', '3', '4', '5', '6', '7', '8', '9', '10', '11'
    , '12']   Score = 0
Offmarks:  []
Free numbers:  [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

red row:  ['2', '3', '4', 'X', '6', '7', '8', '9', '10', '11'
    , '12']   Score = 5
Offmarks:  [5]
Free numbers:  [6, 7, 8, 9, 10, 11, 12]
```

Listing 5: Player class

```
class Player():
  """ Represents a Qwixx player """
  def __init__(self, name):
    self.name = name
    self.rows = {
        'green': ColorRow('green', sort_desc=True),
        'red': ColorRow('red', sort_desc=False),
        'yellow': ColorRow('yellow', sort_desc=False),
        'blue': ColorRow('blue', sort_desc=True)
        }

  def __str__(self):
    return f'''\nPlayer "{self.name}": \r
  {self.tot_points()} points \r
  \rColor rows: \r
  {str(self.rows['green'])}\r
  {str(self.rows['red'])}\r
  {str(self.rows['blue'])}\r
  {str(self.rows['yellow'])}\r
  '''
```

```
  def rolls(self, dicedict):
    for d in dicedict:
      dicedict[d].roll()

  def plays(self, val, color):
    playrow = self.rows[color]
    if val in playrow.markable():
      playrow.mark(val)
      return True
    else:  # cannot play
      return False

  def tot_points(self):
    return sum([card.points() for card in self.rows.values()
   ])
```

Listing 6: Player instance example

```
p = Player('Bob')
print(p)

# Output:
Player "Bob":
  0 points
Color rows:
  green row:  ['12', '11', '10', '9', '8', '7', '6', '5', '4'
    , '3', '2']   Score = 0
  red row:  ['2', '3', '4', '5', '6', '7', '8', '9', '10', '
    11', '12']   Score = 0
  blue row:   ['12', '11', '10', '9', '8', '7', '6', '5', '4'
    , '3', '2']   Score = 0
  yellow row:   ['2', '3', '4', '5', '6', '7', '8', '9', '10'
    , '11', '12']   Score = 0
```

Listing 7: Function for playing a turn

```
def play_turn(dice, idx_active, list_players):
```

```
    """
    Simulate playing a turn of Qwixx.
    Random naive strategy (without looking at the colored rows
     and without passing unless no mark possible)
    """
    # Active player starts
    for p in list_players:
      if list_players.index(p) == idx_active:
        print(f'\nActive player "{p.name}" plays')
        # Active player rolls the all dice
        p.rolls(dice)
        print('Dice:')
        for d in dice:
          print(d, ' ', dice[d].face)

        # Active  player picks 1 color + 1 white dice and marks
       off
        colpick = random.choices(['green', 'red', 'yellow', '
     blue'])[0]
        whipick = random.choices(['white1', 'white2'])[0]
        hit = p.plays(dice[whipick].face + dice[colpick].face,
     colpick)  # sum of 2 dice is played on the same row
        print(p)

    # Other players:
    for p in list_players:
      if list_players.index(p) != idx_active:
        print(f'\nPlayer "{p.name}" plays')
        pick_dice = ['white1', 'white2']
        colpick = random.choices(['green', 'red', 'yellow', '
     blue'])[0]
        hit = p.plays(dice['white1'].face + dice['white2'].face
     , colpick)  # sum of 2 dice is played on the same row
        print(p)

    print('\nEnd of turn'.ljust(80, '.'))

    next_player = (idx_active + 1) % len(list_players)

    return next_player
```

Listing 8: Script for creating players and a set of dice

```
# Create dice set and players for simulating a turn

dice_set = {'white1': Dice(color='white'), 'white2': Dice(
    color='white')}
dice_set.update(dict([(c, Dice(color=c)) for c in ['green', '
    red', 'yellow', 'blue']]))

# Test turn
test_players = [Player('Rock'), Player('Paper'), Player('
    Scissors')]
active_player = 0
```

Listing 9: Game script

```
scores = [p.tot_points() for p in test_players]
k = 0
while max(scores) < 50:
  print(f'\n\nTurn {k+1}\n')
  active_player = play_turn(dice_set, active_player,
   test_players)
  scores = [p.tot_points() for p in test_players]
  k += 1

print(f'\n\nGame over! {k} turns were played.')
```

Listing 10: Playing turn example

```
Turn 1


Active player "Rock" plays
Dice:
white1   3
white2   5
green   4
red   4
yellow   2
blue   1

Player "Rock":
  7 points
```

```
Color rows:
  green row:  ['12', '11', '10', '9', '8', '7', '6', '5', '4'
   , '3', '2']   Score = 0
  red row:  ['2', '3', '4', '5', '6', '7', '8', '9', '10', '
   11', '12']   Score = 0
  blue row:   ['12', '11', '10', '9', '8', '7', '6', '5', '4'
   , '3', '2']   Score = 0
  yellow row:   ['2', '3', '4', '5', '6', 'X', '8', '9', '10'
   , '11', '12']   Score = 7


Player "Paper" plays

Player "Paper":
  8 points
Color rows:
  green row:  ['12', '11', '10', '9', 'X', '7', '6', '5', '4'
   , '3', '2']   Score = 8
  red row:  ['2', '3', '4', '5', '6', '7', '8', '9', '10', '
   11', '12']   Score = 0
  blue row:   ['12', '11', '10', '9', '8', '7', '6', '5', '4'
   , '3', '2']   Score = 0
  yellow row:   ['2', '3', '4', '5', '6', '7', '8', '9', '10'
   , '11', '12']   Score = 0


Player "Scissors" plays

Player "Scissors":
  8 points
Color rows:
  green row:  ['12', '11', '10', '9', 'X', '7', '6', '5', '4'
   , '3', '2']   Score = 8
  red row:  ['2', '3', '4', '5', '6', '7', '8', '9', '10', '
   11', '12']   Score = 0
  blue row:   ['12', '11', '10', '9', '8', '7', '6', '5', '4'
   , '3', '2']   Score = 0
  yellow row:   ['2', '3', '4', '5', '6', '7', '8', '9', '10'
   , '11', '12']   Score = 0


End of turn
   ..............................................................
```

```
Turn 2


Active player "Paper" plays
Dice:
white1   6
white2   4
green   1
red   2
yellow   3
blue   2

Player "Paper":
  13 points
Color rows:
  green row:  ['12', '11', '10', '9', 'X', '7', '6', 'X', '4'
   , '3', '2']   Score = 13
  red row:  ['2', '3', '4', '5', '6', '7', '8', '9', '10', '
   11', '12']   Score = 0
  blue row:   ['12', '11', '10', '9', '8', '7', '6', '5', '4'
   , '3', '2']   Score = 0
  yellow row:   ['2', '3', '4', '5', '6', '7', '8', '9', '10'
   , '11', '12']   Score = 0


Player "Rock" plays

Player "Rock":
  17 points
Color rows:
  green row:  ['12', '11', 'X', '9', '8', '7', '6', '5', '4',
    '3', '2']    Score = 10
  red row:  ['2', '3', '4', '5', '6', '7', '8', '9', '10', '
   11', '12']   Score = 0
  blue row:   ['12', '11', '10', '9', '8', '7', '6', '5', '4'
   , '3', '2']   Score = 0
  yellow row:   ['2', '3', '4', '5', '6', 'X', '8', '9', '10'
   , '11', '12']   Score = 7


Player "Scissors" plays

Player "Scissors":
  8 points
```

```
Color rows:
  green row:  ['12', '11', '10', '9', 'X', '7', '6', '5', '4'
   , '3', '2']   Score = 8
  red row:  ['2', '3', '4', '5', '6', '7', '8', '9', '10', '
   11', '12']   Score = 0
  blue row:   ['12', '11', '10', '9', '8', '7', '6', '5', '4'
   , '3', '2']   Score = 0
  yellow row:   ['2', '3', '4', '5', '6', '7', '8', '9', '10'
   , '11', '12']   Score = 0


End of turn
   ..............................................................




...

Turn 13


Active player "Rock" plays
Dice:
white1   2
white2   6
green   2
red   6
yellow   4
blue   3

Player "Rock":
  37 points
Color rows:
  green row:  ['12', '11', 'X', '9', '8', '7', 'X', '5', '4',
    '3', '2']    Score = 16
  red row:  ['2', '3', '4', '5', '6', '7', 'X', '9', '10', '
   11', '12']   Score = 8
  blue row:   ['12', '11', '10', '9', '8', '7', 'X', '5', '4'
   , '3', '2']   Score = 6
  yellow row:   ['2', '3', '4', '5', '6', 'X', '8', '9', '10'
   , '11', '12']   Score = 7


Player "Paper" plays
```

```
Player "Paper":
  50 points
Color rows:
  green row:  ['12', '11', '10', '9', 'X', '7', '6', 'X', '4'
   , '3', '2']   Score = 13
  red row:  ['2', '3', '4', '5', '6', '7', 'X', '9', '10', '
   11', '12']   Score = 8
  blue row:   ['X', '11', '10', '9', '8', '7', '6', '5', '4',
    '3', '2']    Score = 12
  yellow row:   ['2', '3', '4', '5', 'X', '7', '8', '9', '10'
   , 'X', '12']    Score = 17


Player "Scissors" plays

Player "Scissors":
  43 points
Color rows:
  green row:  ['12', '11', '10', '9', 'X', 'X', '6', '5', '4'
   , '3', '2']   Score = 15
  red row:  ['2', '3', '4', 'X', '6', '7', '8', '9', '10', 'X
   ', '12']    Score = 16
  blue row:   ['12', '11', '10', '9', '8', '7', '6', '5', '4'
   , '3', '2']   Score = 0
  yellow row:   ['2', '3', 'X', '5', '6', '7', 'X', '9', '10'
   , '11', '12']   Score = 12


End of turn
   ..............................................................



Game over! 13 turns were played.
```

Medieval volume measurement system

B.14. [5 points]

Several solutions were possible for the task. The solution provided does the conversion by first determining the equivalent volume in gills and then expresses it in terms of other units. A solution that scored all possible points

needed to have the following features:

- Correctly handles the definition of equivalent volume as given by the task text.
- The conversion algorithm works.
- Correctly interprets the data produced by the conversion algorithm.
- Reproduces the output, including 's' in gallons.
- No major misuse of Python data structures and functions.

Listing 11: Conversion between liters and medieval measurement system.

```
import math

def convert_l_to_UK(liters):
    ls_in_gils=4.546/(2**5)

    #We need the first integer that is larger than (liters/
   ls_in_gils). We get it through math.ceil function.
    ngils=math.ceil(liters/ls_in_gils)
    #An equivalent solution is
    #ngils=int(liters/ls_in_gils)
    #if((liters%ls_in_gils)!=0):
    #    ngils+=1

    units=["gallon", "pottle", "quart", "pint", "chopin", "
   gill"]
    gils_p_unit=32
    print(f'{liters}l fits into:')
    #Now do the conversion to other units by using integer
   operations and starting from larger to smaller units.
    for i, unit in enumerate(units):
        nunits=ngils//gils_p_unit
        if(nunits==1):
            print(f'{nunits} {unit}')
        elif(nunits>1):
            print(f'{nunits} {unit}s')

        #Subtract the ammount contained in the current unit.
        ngils-=nunits*gils_p_unit
        #Change the conversion ratio
        gils_p_unit=gils_p_unit//2
```

Alliterations

B.15. [5 points]

Several solutions were possible for the task. The solution provided counts the alliterations and gives the length of the longest by a single pass through the sentence but other solutions were also possible. A solution that scored all possible points needed to have the following features:

- It works for capitalized words.
- The algorithm works correctly to determine the number of alliterations.
- The algorithm works correctly to determine the length of the longest alliteration.
- Return values are correctly extracted from the data structures used to analyze the alliterations.
- No major misuse of Python data structures and functions.

Listing 12: One pass solution for alliteration problem.

```
def analyze_alliterations(sentence):
    # Preprocess the input
    words=[part.lower() for part in sentence.split()]

    #Set up the initial conditions for the analysis

    #Initial store first character of the first word, since
   the loop starts working from the second word on
    char_init=words[0][0]

    #Counts the number of words in an alliteration,
   initialized to zero.
    #If it is larger than zero it means that we are inside an
    alliteration.
    counter=0

    #Counts the number of alliterations found in the sentence
    n_allit=0

    #Records the length of the longest alliteration
   encountered so far
    n_words=0
```

```
    #We start checking for alliterations from the second word
    onward
    for i in range(1, len(words)):
        if(char_init==words[i][0]): # this is an alliteration
            if(counter==0): #it is new to us
                n_allit+=1
                counter=2 #when we first see an alliteration
    it has length 2 since it consists of 2 words
            else:
                counter+=1
        else: #this is not an alliteration
            char_init=words[i][0] #remember the new first
    letter to test against it later
            n_words=max(n_words, counter)
            counter=0 #set counter to zero, next repetition
    of initial characters will be a new allteration

    #If the sentence ends with an alliteration, we need to
    check if it is maybe the longest we have ever seen
    n_words=max(n_words, counter)

    return n_allit, n_words
```