Suggested solutions and comments for the 2021-03-12 exam in DA2004/DA2005

Part A: multiple choice questions (1p per question)

1. *B, E*
2. *D*
3. *A, C*
4. *C, D*
5. *E*
6. *A, C, E*
7. *E*
8. *E*

Part B: coding problems (2p per problem)

9. Possible solutions:

```
import random

class Dice:

    def roll_die(self):
        return random.randint(1,6)

    def roll_dice(self,n):
        sum = 0
        for i in range(n):
            sum += self.roll_die()
        return sum

d = Dice()
print(d.roll_die())
print(d.roll_dice(2))
print(d.roll_dice(20))
```

10. Possible solution:

```
def roll_two_dice(n_times):
    d = Dice()
    if type(n_times) == int and 0 < n_times:
        return [ d.roll_dice(2) for i in range(n_times) ]
    else:
        raise ValueError("n_times must be a positive integer")

print(roll_two_dice(10))
```

11. Possible solution:

```
def result_of_two_dice_rolls(n_times):
    xs = roll_two_dice(n_times)

    for i in range(2,13):
        print(i,format(xs.count(i)/n_times * 100,'.1f')+"%")

result_of_two_dice_rolls(1000)
```

12. Possible solution:

```
mydict = { 'hej' : 12 , 'hopp' : 10101 , 'a' : 1 }

def print_right_aligned(d):
    max_len_keys = max(map(len,d.keys()))
    max_len_vals = max(map(lambda x: len(str(x)),d.values()))

    for k,v in d.items():
        sv = str(v)
        print((max_len_keys - len(k)) * ' ' + k
             ,(max_len_vals - len(sv)) * ' ' + sv)

print_right_aligned(mydict)
```

13. Possible solution:

```
def best_corner(board):
    n = len(board)
    best_square = ()
    best_score = 0
    for i in range(1, n):
        for j in range(1, n):
            score = 0
            for k1 in range(i-1, i+1):
                for k2 in range(j-1, j+1):
                    score += board[k1][k2]

            if score > best_score:
                best_score = score
                best_square = (i,j)
    return best_square, best_score
```

Notice that the outer loops iterate over the inner corners thanks to `range(1, n)`. We access the squares around a corner using the inner two loops.

The solution might be nicer and easier to read if the score computation is broken out to a helper function:

```
def best_corner(board):
    n = len(board)
    best_square = ()
    best_score = 0
    for i in range(1, n):
        for j in range(1, n):
            score = compute_score(board, i, j)
            if score > best_score:
                best_score = score
                best_square = (i, j)
    return best_square, best_score


def compute_score(board, i, j):
    n = len(board)
    score = 0
    for k1 in range(i-1, i+1):
        for k2 in range(j-1, j+1):
            score += board[k1][k2]
    return score
```

This might make it more clear the the *i* and *j* loops are over coordinates, and the *k* loops in `compute_score` are over squares?

Since these two functions are so co-dependent, some prefer to make the helper function as an *internal* function, which is only in scope for `best_corner` and not visible to other functions:

```
def best_corner(board):
    def compute_score(board, i, j):
        n = len(board)
        score = 0
        for k1 in range(i-1, i+1):
            for k2 in range(j-1, j+1):
                score += board[k1][k2]
```

```
    n = len(board)
    best_square = ()
    best_score = 0
    for i in range(1, n):
        for j in range(1, n):
            score = compute_score(board, i, j)
            if score >= best_score:
                best_score = score
                best_square = (i, j)
    return best_square, best_score
```

Which solution do you prefer?

14. Possible solution:

```
class Board:
    def __init__(self, board, occupied_positions=[]):
        self.board = board
        self.board_size = len(board)
        self.occupied_positions = occupied_positions

    def best_valid_corner(self):
        best_square = ()
        best_score = 0

        # Adding 1 to max coordinate because more corners
        # than squares. We are looping over corners here.
        for i in range(self.board_size + 1):
            for j in range(self.board_size + 1):
                if not (i, j) in self.occupied_positions:
                    score = self.compute_score(i, j)
                    if score >= best_score:
                        best_score = score
                        best_square = (i, j)
        return best_square

    def compute_score(self, i, j):
        score = 0
        for k1 in range(max(i-1, 0), min(i+1, self.board_size)):
            for k2 in range(max(j-1, 0), min(j+1, self.board_size)):
                score += self.board[k1][k2]
        return score
```

Notice that `compute_score` now handles that coordinates are not constrained to the inner corners. We guarantee valid indices for the squares with min and max, no matter which corner coordinates are given in the call.