MATEMATISKA INSTITUTIONEN
STOCKHOLMS UNIVERSITET
Avd. Beräkningsmatematik
Kursledare: Anders Mörtberg
Examinator: Lars Arvestad

Tentamensskrivning i
DA2005 Programming techniques
7.5 hp
2021-03-12

- Part A has multi-choice questions with at least one correct answer. The wrong answer or the wrong number of answers both give zero points.
- You need to pass Part A (4 correct answers on 8 questions) for your Part B to be graded.
- Part B is a number of problems worth a total of 12 points, which are to be solved using Python 3 code.
- The answers to Part B are handed in as a single `.py` file named like `anonymouscode.py` where `anonymouscode` is the code given to you when registering for the exam in Ladok. You **must** also write your personal code in a comment at the top of your Python file. You **must not** write your own name anywhere in the file!
- Carefully use the identifiers for functions, methods, and variables as requested in the problems.
- No `import` statements may be used unless mentioned and requested in the problem. You are free to use any functions defined in the Python standard environment (available at startup), including `len`, `range` and `map`.
- You should write **Python 3** code, and *not* Python 2.7, for example.
- **Resources:** You are allowed a "cheat sheet" for Part A: an A4 paper filled with as much information as you like. It can be written/printed on both sides. Part B is *open book*, so the same rules apply for Part B as in labs and project.
- **Grading thresholds:** E: 10, D: 12, C: 14, B: 16, A: 18, out of maximum 20.

---

## Part A: multi-choice questions (1p per question)

**1**. Which of the following words are specifically about error handling?

A. `with`
B. `raise`
C. `class`
D. `close`
E. `except`

**2**. If `mylist = ['hej', 1, 5, True, 3, 9]`, then what is `mylist[-3]`?

A. `None`
B. `5`
C. `3`
D. `True`
E. None of the above, because you cannot use negative index for lists.

**3**. Which of the following is/are reserved words in Python?

A. `class`
B. `instance`
C. `if`
D. `print`
E. `__init__`

**4**. What claims are true?

A. Lists are immutable.
B. `(1,2,3)` is a list with three elements.
C. Strings can be *keys* in a `dict`.
D. Strings can be *values* in en `dict`.
E. There is a type `char` for characters in Python.

**5**. What is the result of `list(filter(lambda s: s[0].lower()=='a', ['Hej', 'apan', 'heter', 'Anders']))`?

A. `[]`
B. `['Hej', 'apan', 'heter', 'anders']`
C. `['apan']`
D. `['apan', 'anders']`
E. `['apan', 'Anders']`

**6**. Given the function below, which alternatives returns `True`?

A. `f(True)`
B. `f(y=False)`
C. `f(False,False)`
D. `f(True,False)`
E. `f()`

```
def f(x=True,y=True):
    return ((x and y) or (not x and not y))
```

**7**. Given the code below, which alternatives results in an exception?

```
class A:
    x1 = 1

    def f1(self):
        return "f1 of A"

class B(A):
    def f2(self):
        x2 = 3
        return "f2 of B"

class C(B):
    def f3(self):
        return "f3 of C"

c = C()
```

A. `print(c.f1())`
B. `print(c.f2())`
C. `print(c.f3())`
D. `print(c.x1)`
E. `print(c.x2)`

**8**. What is printed by the code on the right?

A. `[1, 2, 3]`
B. `[]`
C. `[1, 2, 3, 3, 2, 1]`
D. `[-1, -2, -3]`
E. `[3, 2, 1]`

```
def f(mylist):
    if mylist != []:
        x , t = mylist[-1] , mylist[:-1]
        return [x] + f(t)
    else:
        return mylist

print(f([1,2,3]))
```

## Part B: coding problems (2p per problem)

**Note**: you may use the `random` module in this part, so your solution file can contain `import random`.

9. Implement a class `Dice` that represents six-sided dice and their outcomes. The class should have two methods:

   - `roll_die()`: returns the outcome from a dice roll
   - `roll_dice(n)`: returns the outcome (i.e., the sum) from rolling `n` dice.

   **Tests:**

   ```
   [In] : d = Dice()
   [In] : print(d.roll_die())
   [Out]: 2
   [In] : print(d.roll_dice(2))
   [Out]: 7
   [In] : print(d.roll_dice(20))
   [Out]: 74
   ```

   **Note:** as dice rolls are random, you are likely to get other outcomes when trying the tests.

10. Write a function `roll_two_dice(n_times)` that takes a positive number `n_times` (i.e., `n_times > 0`), and with the help of list comprehension returns a list with the outcomes of `n_times` dice rolls (using the `Dice` class).

    **Note:** for full credit, you must use the `Dice` class and a list comprehension. If `n_times` is not a positive number, then an exception should be raised by the function.

    **Tests:**

    ```
    [In] : print(roll_two_dice(10))
    [Out]: [9, 2, 7, 2, 9, 7, 8, 10, 3, 5]
    ```

    **Notes:** since dice rolls are random, you may get other outcomes when testing.

11. Write a function `result_of_dice_rolls(n)` that takes a positive number `n` and presents the result of throws with two dice (using the `Dice` class) in a table. Each line shows one outcome and, in percent, what share of rolls the outcome corresponds to.

    **Hint:** use `roll_two_dice` from the problem above.

    **Example output:**

    ```
    2 2.4%
    3 6.1%
    4 7.9%
    5 11.6%
    6 13.5%
    7 16.7%
    8 14.0%
    9 10.3%
    10 8.9%
    11 6.5%
    12 2.1%
    ```

    **Note:** since the result is random, you will get different output than the example above.

12. Given a dictionary `d` from strings to numbers, write a function `print_right_aligned(d)` that prints a table with key/value pairs on each line Both keys and values should be right-aligned.

    For example, `{ 'hej': 12 , 'hopp': 10101 , 'a': 1 }` should be printed like:

    ```
     hej    12
    hopp 10101
       a     1
    ```

**13.** We define a board game over boards with $k \times k$ squares, $k > 1$. Every square has points, which are given as nested lists ($k$ lists of length $k$). You put down markers on the square *corners* and get a score as the sum of the points for the 1–-4 squares a marker is on. The corners are indexed from 0 to $k$, see Figure 1!

Write a function `best_corner(b)` that returns the best corner on board $b$ to put the first marker on, and the score you get from that corner. If there are several corners with the same score, then it does not matter which of them you return.

*Hint:* It is sufficient to only consider the inner corners of the board, all covering four squares.

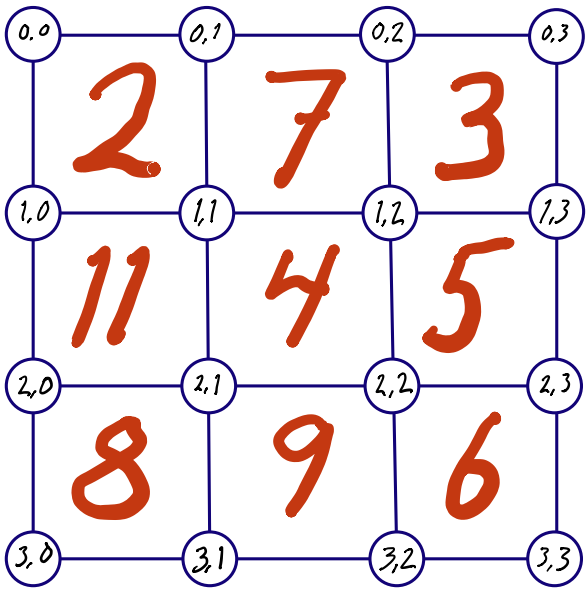


Figure 1: Board for problem 13 corresponding to `[[2, 7, 3], [11, 4, 5], [8, 9, 6]]`. Coordinates are shown with black number pairs in the corners and square points are given with red numbers. The best placement of a marker is coordinate `(2, 1)` since it yields score $11 + 4 + 8 + 9 = 32$.

**Tests:**

```
[In] : board0 = [[0, 0], [0, 1]]
[In] : corner, score = best_corner(board0)
[In] : print(corner, score)
[Out]: (1, 1) 1
[In] : board1 = [[10, 1, 0], [1, 1, 0], [0, 0, 0]]
[In] : corner, score = best_corner(board1)
[In] : print(corner, score)
[Out]: (1, 1) 13
[In] : board2 = [[0, 0, 0], [0, 1, 10], [0, 1, 1]]
[In] : corner, score = best_corner(board2)
[In] : print(corner, score)
[Out]: (2, 2) 13
[In] : print(best_corner([[2, 7, 3], [11, 4, 5], [8, 9, 6]]))
[Out]: ((2,1), 32)
```

**14.** Implement a class `Board` for the board game in problem 13. The class should have a constructor with parameters for both board and a list of already chosen positions, because that information should be stored in the objects. Add a method `best_valid_corner` that returns the best-scoring corner, while avoiding the corners already chosen (i.e., do not return a position where there is already a marker).

You can assume that this method is not called when all corners are already taken.

*Hint:* You cannot limit yourself to the inner corners, because they may be taken already.

**Tests:**

```
board0 = [[0,0],[0,1]]
board1 = [[10,1,0], [1, 1, 0],[0,0,0]]
board2 = [[0,0,0], [0,1, 10], [0, 1, 1]]
board3 = [[2,7,3],[11, 4, 5],[8, 9, 6]]

b0 = Board(board0, [])
b1 = Board(board1, [])
b2 = Board(board2, [])

assert b0.best_valid_corner() == (1, 1)
assert b1.best_valid_corner() == (1, 1)
assert b2.best_valid_corner() == (2, 2)

b0 = Board(board0, [(1,1)])
b1 = Board(board1, [(1,1)])
b2 = Board(board2, [(2,1), (2,2)])

assert b0.best_valid_corner() in [(1, 2), (2, 1), (2, 2)]
assert b1.best_valid_corner() in [(0, 1), (1, 0)]
assert b2.best_valid_corner() in [(1, 2), (2,3)]
```