Sensorveiledning INF-1049 høst 2023 ordinær eksamen

Generelt er syntaks ikke så viktig. De har ikke tilgang til noen god editor på eksamen og kan heller ikke kjøre programmer for å luke ut feil. Det vi i all hovedsak er ute etter er logikken, og at de tenker riktig og bruker konseptene fra pensum riktig.

I alle oppgaver kan de lage ekstra-metoder og -klasser der de ønsker det selv. Det skal dog ikke være nødvendig gitt oppgaven, alt er løselig uten dette, men det trekkes ikke for det.

Oppgave 1

Oppgave 1.1 (3%)

```
t = (1, 2, 3, 4)
t[2] = "3"

print(t)
```

Gitt koden over. Hva blir resultatet?

C. Det gir en feil, tuples er immutable og kan ikke endres når de er opprettet.

Oppgave 1.2 (3%)

```
l = [x for x in range(2, 10, 2)]
```

Hvordan vil listen se ut?

B. `[2, 4, 6, 8]`

Oppgave 1.3 (3%)

```
with open("data.csv", "w+") as fp:
    for line in fp:
        print(line)
```

Hva er sant om *mode* argumentet til *open?*

A. **a** lar oss legge til data på slutten av filen.

Oppgave 1.4 (3%)

```
triangle = {
    "a": (2, 3),
    "b": (6, 2),
    "c": (1, 3)
}
```

Hva er sant om dictionaries?

D. Både **tuples** og **strings** kan være *keys.*

Oppgave 1.5 (3%)

```
class Database:
    def __init__(self, data_dict=None):
        self.data = data_dict
```

Hva er *self* argumentet i en klassemetode?

A. **self** referer til objektet selv, og trenger ikke å sendes med som argument.

Oppgave 2

Under er hovedmomentene i oppgavene. Om noe mangler så kan man likevel få noe poenguttelling for delvis korrekte svar.

2.1 En klasse inneholder metoder som opererer på dataene som er lagret i objekter. Inneholder instansvariabler. En dictionary inneholder kun data som er kodet nøkkel:data.
2.2 Name-main-blokka lar oss teste koden vår inne i en python-fil/modul uten at den kjøres når vi importerer modulen i en annen python-fil.

2.3 Metoder som starter og slutter med doble understreker angir spesiell oppførsel i klasser. Man har metoder som overskriver operatorer (+, - osv), og metoder slik som str, som lar oss få en tekstlig representasjon av et objekt, heller enn en minneadresse.
2.4 En logisk feil er en feil programmerer har gjort som fører til feil resultat. Typisk sier vi at en kjøretidsfeil fører til at programmet krasjer, mens en logisk feil gir feil resultat fra et kjørbart program.
2.5 Studenten bør ha en funksjon som inneholder en løkke som går **fra** parameteren-1 og ned til 1. Inne i denne en if-sjekk som undersøker om tallet er delelig vha. modulo og eventuelt returnerer False. Om den når ned til 1 i løkka returneres True etter løkka.

Oppgave 3

I oppgave 3.1 er det brukt en liste, det kan man lese ved å se at man bruker append og remove i metodene. Fordelen her ved å bruke en liste (fremfor f.eks. dictionary) er at vi kun lagrer strenger, og det ikke er noen key:value pairs. Det er også enkelt å løkke gjennom en liste, slik vi ser i for-løkken i str-metoden.

I oppgave 3.2 finnes den mest opplagte feilen i flytt_kommune. Om en kommune ikke finnes i listen, vil det ikke være mulig å fjerne den, og vi får en error. Man kan løse dette på flere måter, den enkleste er antakelig en if-setning i starten av metoden som sjekker om kommunen allerede finnes i lista. Man kan også raise en exception, typisk da ValueError.

I oppgave 3.3 skal studenten ha med metoden __add__(self, other). Denne bygger opp et nytt objekt av klassen Fylke og gir dem instansvariabler med korrekte verdier. Det trekkes mye om metoden ikke returnerer et nytt objekt. Det trekkes litt ved feil verdier for nye instansvariabler. Hovedpoenget er at de skal se at man lager et nytt fylkesobjekt ved bruk av operatoren +.

Oppgave 4

Oppgave 4.1

Her skal kandidaten bestemme seg for en datatype, og implementere en funksjon for å sette inn brikker i brettet i vårt 3-på-rad spill. For å få full uttelling så bør kandidaten ha diskutert følgende:
Fornuftige datatype kan da være strings, hvor man skiller spillerne på bokstav. Dette er enkelt å sjekke, og det er enkelt å evt legge til flere spillere senere, som gjør str egnet.
Det er også mulig å bruke int, som kan kombineres senere med når man sjekker 3 på rad. Her er det også mulig å f eks bruke negative tall for en spiller, og positive tall for en annen.
Funksjonen kan implementeres på følgende måte i Python, evt noe lignende:

```
def place_piece(x, y, player, board):

    if y > len(board):
        raise IndexError

    if x > len(board[0]):
        raise IndexError
```

```
    if board[x][y] != 0:
        raise ValueError

    board[x][y] = player
```

Det trenger ikke å implementeres i korrekt Python kode, hvis det er gitt Pseudokode som oppnår det samme så skal dette også gi full uttelling.
Typen error som returneres bør gi mening, men her er det flere som passer. F eks **ValueError** og **IndexError.** Hvis typen error ikke gir helt mening, eller at det er returnert istede for *raised* så trekker dette litt ned.

Oppgave 4.2

Å sjekke brettet kan gjøres på flere måter. Så lenge kandidaten klarer å sjekke om noen har vunnet spillet og returnere det som funksjonen skal (True, player) hvor *player* er den datatypen de bestemte seg for i forrige oppgave, så skal det gi full uttelling.
Ting som trekker ned er: Veldig mye unødig eller rotete kode, kompliserte løsninger, oppnår ikke det oppgaven spør etter (f eks sjekker bare en retning) eller at koden inneholder feil.
Koden trenger ikke å være 100% korrekt for å få full uttelling, hvis kandidaten har de riktige ideene og har bare slitt litt med å implementere det i Python, så skal det ikke trekke noe, og er det noen feil i tankegangen så trekker det litt ned. De kan også skrive Pseudokode om de ønsker, så det er ikke krav om at det er Python kode.
Koden kan implementeres slik:

```
def check_player(board, player, dir):

    piece_count = 0

    for i in range(len(board)):
        for k in range(len(board[0])):

            if board[i][k] == player and dir == "hor":
                    piece_count += 1
            elif board[k][i] == player and dir == "vert":
                    piece_count += 1
            else:
                    piece_count = 0

            if piece_count >= 3:
                return True

    return False


def check_board(board):


    if check_player(board, "B", "hor")\
    or check_player(board, "B", "vert"):
```

```
        return True, "B"

    if check_player(board, "R", "hor")\
    or check_player(board, "R", "vert"):

        return True, "R"

    return False, None
```

Her er det også brukt hjelpefunksjon for å gjøre ting litt mer lesbart.

Oppgave 4.3

I denne oppgaven så trenger ikke kandidaten å implementere noe, men heller diskutere og identifisere problemene med større brett, og eventuelle løsninger på disse problemene i forhold til den tidligere koden som er skrevet.
De to punktene oppgaven nevner er *størrelsen på brettet* og *tiden det tar å sjekke.* Her bør kandidaten peke på:

- Størrelsen på matrisen vil ta opp mye minne.
- Mange av plassene vil være 0, og derfor ikke inneholde viktig informasjon (sparse matrise, men ikke påkrevd at de identifiserer det som dette)
- Å sjekke for *n* på rad med stor *n* kan være ineffektivt når man må gjøre dette hver eneste gang det settes inn en ny brikke.

Noen løsninger på disse problemene kan da være:

- Å kun lagre de trekkene som faktisk blir gjort, slik at brettet egentlig bare har en bredde og høyde, men at man faktisk ikke lagrer noe annen data enn de brikkene man setter inn. Dette gjør at man underveis kan sjekke for en vinner eller ikke (at man holder styr på en vinnerliste f eks, som fylles og tømmes som nødvendig).
- Om man har en fullstørrelse matrise, så kan man gjøre noen optimaliseringer når man sjekker. Hvis man begynner å sjekke en spiller, men ikke finner n-på-rad, så kan man hoppe n-1 posisjoner før man begynner å sjekke igjen.
- Man kan også bruke et library som støtter sparse matriser. Og effektive operasjoner på disse.

Her er det mange løsninger og problemer, og mye kandidaten kan diskutere. Det trenger ikke å være nøyaktig disse punktene over, men det bør være noe relatert til ytelse.