Sensorveiledning

INF-1049 H22

Oppgave 1

Deloppgavene er automatisk rettet av Wiseflow. Svarene står under.

Oppgave 1.1

True **AND** True - True

True **AND** False - False

False **OR** False – False

True **OR** False – True

NOT True – False

Oppgave 1.2

Hva er sant om forskjellen mellom en *liste* og en *tuple* i Python?

- Lister kan forandres etter de er opprettet, det kan ikke tupler.

Oppgave 1.3

```
text = "Nørge"
text[1] = "o"
```

Er det mulig å gjøre som koden over i Python?

- Nei, strings er *immutable* i Python.

Oppgave 1.4

```
var = 500

def scoped(var):
    return var + 20

res = scoped(200)
```

Hva blir verdien til *res?*

- 220, fordi *var* er lokal inne i funksjonen og tar verdien til parameteret.

Oppgave 1.5

```
class MyClass:
    def __init__(self, initial):
        self.val = initial

    def __add__(self, other):
        return MyClass(self.val + other.val)

    def __sub__(self, other):
        return MyClass(self.val - other.val)
```

Hva er formålet med *double underscore (dunderscore) methods* i klasser?

- Definerer oppførselen til en klasse når man bruker symboler som +, -, *, / etc på objekter av klassen.

Oppgave 2

Deloppgavene er besvart med korte tekst spørsmål. Svarene her er forslag til svar, som tar for seg essensen i spørsmålet, men er ikke de *eneste* korrekte svarene. Kandidater som utgreier og gir fyldigere svar får gjerne full uttelling.

Oppgave 2.1

Forklar kort hvordan du ville representert en *2-dimensjonal matrise* i Python (uten bruk av moduler og libraries).

Beskriv også hvordan du leser, sletter og legger til verdier i en slik matrise.

- Her er det flere muligheter, men den mest åpenbare er å bruke en liste av lister. Hvor hver sub-liste da representerer en rad i matrisen (evt en kolonne). For å lese, så bruker man doble firkant-parenteser `list[row][col]`. For å slette og legge til verdier så må man bestemme seg for

om man matrisen skal være fast, og man sletter/legger til hele rader. Evt så kan man bruke None for å representere enkeltverdier som er slettet fra matrisen. Å legge til dimensjoner kan gjøres med append, men da er det viktig å appende alle dimensjonene for å opprettholde matrisen.

Oppgave 2.2

Hvorfor bruker vi *klasser* i Python, i stedet for å alltid bruke *dictionaries*?

Hvilke fordeler og ulemper har man ved å bruke *klasser*?

- I utgangspunktet, så er det ikke noe som forhindrer oss fra å bruke kun *dictionaries* og *funksjoner.* Men dette blir fort uoversiktlig. Den største fordelen med klasser er at man kan kombinere atributter og metoder som er relatert til hverandre. Man kan også modifisere oppførsel med *dunderscore* metoder som forenkler bruken av objektene etterpå. En ulempe med å bruke klasser er at mye blir skjult for brukeren. Overdreven bruk av klasser øker også mengden kode som skrives, som kan virke mot sin hensikt med lesbarhet.

Oppgave 2.3

Hva er hensikten med å bruke konstruksjonen ***if __name__ == "__main__"***?

- Hensikten er å forhindre at kode kjøres når man importerer filer som moduler. Den gjør at kun den filen som kjøres med *python file.py* får navnet __main__ og kjører koden som står etter konstruksjonen.

Oppgave 2.4

Hva er forskjellen på en *runtime error (kjøretidsfeil)* og en *logisk feil*?

- En *runtime error* betyr at vi har gjort noe som Python ikke tillater. F eks dele på null, bruke variabler som ikke er opprettet, prøver å hente ut en key som ikke finnes i en dict. Da vil programmet si ifra/krasje med en error. En *logisk feil* er noe galt i algoritmen som er implementert. Det er ingen ting galt med koden i seg selv, men vi får likevel feil resultat fordi vi har tenkt feil når vi implementerte. *Logiske feil* er veldig mye vanskeligere å finne en *kjøretidsfeil.*

Oppgave 2.5

Du skal evaluere en funksjon over intervallet [-3, 3] med steglengde 0.1. Forklar hvorfor du ikke kan bruke en for-løkke og foreslå et alternativ som fungerer.

- En for-løkke kan ikke jobbe med flyttall, bare med heltall (gjennom range). Vi kan i stedet benytte en while-løkke med en float variabel som vi oppdaterer med steglengden hver iterasjon. Eventuelt så kan man bruke Numpy sin linspace for å generere alle tallene på forhånd og bruke denne listen med en for-loop.

Oppgave 3

Her skal kandidatene lese og forstå kode, og deretter bruke denne kunnskapen til å videreutvikle koden. Det er flere måter å gjøre dette på, så her vil det kun være forslag til løsning. Der kandidatene skal utvide koden så trenger de ikke gjenta den oppgitte koden. Hovedpoenget er at kandidatene har forstått hvordan å bruke klassen som er gitt:

```
class Species:

    def __init__(self, name, avg_weight, avg_height, main_feature):
        self.name = name
        self.weight = avg_weight # in kg
        self.height = avg_height # in meters
        self.features = {"Main_feature": main_feature}

    def add_feature(self, feature_type, feature_name):
        self.features[feature_type] = feature_name

    def get_main_feature(self):
        return self.features["main_feature"]

    def body_mass(self):
        return self.weight / self.height**2

    def make_hybrid(self, other):
        new_name = self.name + "x" + other.name
        new_weight = (self.weight + other.weight) / 2
        new_height = (self.height + other.height) / 2
        feature_combination = [self.features["Main_feature"], other.features["Main_feature"]]
        return Species(new_name, new_weight, new_height, feature_combination)

    def __eq__(self, other):
        this_size = self.body_mass()
        other_size = other.body_mass()
        return this_size == other_size

    def __str__(self):
        data = self.name
        data += " with weight " + str(self.weight)
        data += " and height " + str(self.height)
        data += " gives body mass " + str(self.body_mass())
        return data
```

Oppgave 3.1

Klassen inneholder to metoder med navn som starter og slutter med doble understreker.

1. Hva gjør disse to metodene?
2. Hvorfor har metodene doble understreker, og hvilke handlinger tillater metodene oss å gjøre med objekter av klassen?

- __eq__ sammenligner body_mass til dette objektet, med et annet objekt av samme type (Species). __str__ gir en lesbar streng-representasjon av objektet.
- Disse metodene har doble understreker fordi de forandrer oppførselen til objektet i bruk med spesifikke symboler. Dvs __eq__ gjør at man kan sammenligne 2 objekter av Species med

SpieciesA == SpieciesB, som da vil kalle eq metoden. __str__ kalles når vi gjør print(Species) med et objekt av species typen.

Oppgave 3.2

I tre av metodene kan det oppstå kjøretidsfeil. Identifiser disse og forklar hvordan du ville gått frem for å hindre feil i å oppstå.

- I get_main_feature så er “main_feature” stavet feil, og vil ikke eksistere i dictionarien og vi får en KeyError.
- I body_mass, så kan høyden være satt til 0, og vi får en ZeroDivisionError.
- __str__ kan også gi følgefeil fordi den kaller på self.body_mass(), som kan gi en ZeroDivisionError.

Oppgave 3.3

Gitt koden under:

```
if __name__ == "__main__":
    salmon = Species("Salmon", 3, 1.1, "anadromous fish")
    rainbow_trout = Species("Rainbow trout", 5, 1.6, "farmed fish")

    salmon_trout = salmon.make_hybrid(rainbow_trout)
```

Hvilke verdier vil *name*, *weight*, *height* og *main_feature* få i objektet lagret i variabelen *salmon_trout*? Du kan bruke kalkulator ved å trykke på knappen "Calculator" under.

- *name* = ‘SalmonxRainbow trout’
- *weight = 4*
- *height = 1.35*
- *main_feature = {‘Main_feature’: [‘anadromous fish’, ‘farmed fish’]}*

Oppgave 3.4

```python
class Species:
    def __init__(self, name, avg_weight, avg_height, main_feature):
        self.name = name
        self.weight = avg_weight # in kg
        self.height = avg_height # in meters
        self.features = {"Main_feature": main_feature}

        self.subspecies = {}

    def body_mass(self):
        return self.weight / self.height ** 2

    def add_subspecies(self, name, avg_weight, avg_height, main_feature):
        self.subspecies[name] = Species(name, avg_weight, avg_height, main_feature)

    def __str__(self):
        rep = f"{self.name} with weight {self.weight} and {self.height} gives body mass {self.body_mass()}\n"
        rep += "Subspecies:\n"
        for name, s in self.subspecies.items():
            rep += f"{name} with weight {s.weight} and {s.height} gives body mass {s.body_mass()}\n"

        return rep
```

Oppgave 4

Her skal kandidaten programmere selv, basert på oppgitte formler og algoritmer. Det trenger ikke å være korrekt python kode, men pseudokoden bør være så detaljert at man forstår tankegangen. Kandidaten gies poeng basert på om koden løser det oppgaven spør etter. Det trekkes litt poeng om koden er lite lesbar og om kandidaten har bare kopiert formelen direkte, eller om det faktisk er gjort lesbart.

Oppgave 4.1

Implementer en funksjon *dist(city1, city2)* som regner avstanden d

mellom to byer basert på *haversine* formelen:

$h = \sin^2(\varphi_2 - \varphi_1^2) + \cos(\varphi_1) \times \cos(\varphi_2) \times \sin^2(\lambda_2 - \lambda_1^2)$

$d = 2r \times \arcsin(h\sqrt{})$

Hvor φ_1, φ_2

er *latitude* og λ_1, λ_2 er *longitude*. $r = 6372.8$

jordas radius i km.

Du kan anta i alle oppgavene at du har tilgang til funksjoner i python sitt Math bibliotek som sin(x), cos(x), asin(x) (invers sinus), acos(x) (invers cosinus), radians(x) (konverterer grader til radianer) etc.

- Det er ikke nødvendig at kandidatene har konvertert til radianer, da dette ikke var gitt i oppgaven. Men i utgangspunktet så fungerer math sin trigonometriske funksjoner med radianer og ikke grader.

```
def dist(city1, city2):

    # Coordinate distance in radians
    dlat = radians(city1[0] - city2[0])
    dlon = radians(city1[1] - city2[1])

    # Latitude coordinates in radians
    lat1 = radians(city1[0])
    lat2 = radians(city2[0])

    h = sin(dlat/2)**2 + cos(lat1)*cos(lat2)*sin(dlon/2)**2
    d = R * 2 * asin(sqrt(h))

    return d
```

Oppgave 4.2

Vi er en omreisende selger, som skal besøke et sett med byer og vi skal finne korteste mulige rute. Vi skal besøke alle byene *kun* en gang, og vi skal returnere tilbake til startbyen. Startbyen skal være en av byene fra listen. Listen er som følger:

```
cities = [
(69.6652, 18.9193, "Tromsø"),
(63.4340, 10.3628, "Trondheim"),
(46.9539, 7.4235,  "Bern"),
(48.5691, 7.6920,  "Strasbourg"),
(56.9718, 23.9889, "Riga"),
(48.2207, 16.3100, "Wien"),
(52.5069, 13.2846, "Berlin")
]
```

For å finne en eksakt løsning på problemet (*traveling salesman problem*) så må vi prøve alle permutasjoner (ruter) av byene, og se hvilken som er kortest. Algoritmen vi må implementere er da:

```
1. Sett nåværende rute til å være den korteste.
2. Lag en ny permutasjon (rute) av alle byene.
3. Regn ut lengden på permutasjonen.
4. Sjekk om lengden er kortere enn den nåværende, og oppdater den hvis ja.
5. Gjenta fra steg 2 til vi har regnet alle permutasjonene.
```

Implementer to funksjoner:

`path_length(path)` - Regner lengden av en rute fra en liste med koordinater. Husk at du kan bruke funksjonen du laget i 4.1.
`tsp_brute_force(cities)` - Finner en eksakt løsning på traveling salesman problem med algoritmen over.

Hint: For å lage alle permutasjoner av en liste med koordinater, så kan man loope over itertools.permutations(list). Hver runde i loopen vil gi en ny permutasjon av list parameteret.

```python
def tsp_solve_brute_force(cities):

    min_path = cities
    min_length = path_distance(cities)


    # Choose a random permutation of cities
    # Until we find the best solution
    for path in itertools.permutations(cities):
        length = path_distance(path)
        if length < min_length:
            min_length = length
            min_path = path


    return min_path, min_length
```

Oppgave 4.3

I oppgave 4.2 så skulle du finne en eksakt løsning på traveling salesman problem. Dette fungerer veldig bra så lenge vi ikke skal besøke så veldig mange byer, men dette kan fort bli vanskelig når antallet byer øker.

Her skal du ikke skrive kode, men diskuter hva som skjer når vi prøver å finne en eksakt løsning for veldig mange byer (tusenvis og flere). Samt foreslå hvordan vi kan finne en tilnærmet løsning på en mer effektiv måte.

- Her er det viktig at kandidaten forstår og nevner at det å permutere byer som vi gjorde i forrige oppgave er veldig tidkrevende. Når vi prøver å finne en eksakt løsning så ender vi fort opp med n! permutasjoner, så antall og kjøretid øker veldig fort.
- En tilnærmet løsning vil da kunne oppnås med å bruke *nearest neighbor,* en ved å alltid velge den nærmeste byen som neste by.
- Om kandidaten kan diskutere seg frem til andre løsninger som vil være mer effektiv en brute force så gir det også uttelling.

Karakterskala

A	89
B	77
C	65
D	53
E	41
F	0