UiT

NORGES
ARKTISKE
UNIVERSITET

SENSORVEILEDNING

For eksamen i: **INF-1400 Objektorientert programmering**

Dato: **Tirsdag 22. mai 2018**

Sensorveiledningen er på 10 sider inklusiv forside

Fagperson/intern sensor: Lars Brenna

Telefon: 90786723

Eksamen INF-1400
Objektorientert programmering
Vår 2018

Løsningsforslag

Eksamenssettet består av to deler, totalt fire oppgaver.

Les oppgaveteksten grundig og disponer tiden slik at du får tid til å svare på alle oppgavene. I noen oppgaver kan det være nødvendig å tolke oppgaveteksten ved å gjøre noen antagelser - gjør i så fall rede for hvilke antagelser du har gjort, men pass på å ikke gjøre antagelser som trivialiserer oppgaven.

Merk: Fokus i oppgaven er objektorientering og bruk av datastrukturer vi bygger opp. Det gis ikke ekstra poeng for å utvide oppgaven utover det som er spesifisert.

Pseudokode er i de fleste tilfeller godtatt, det kreves ikke kjørbar kode. Python3-syntaks er foretrukket, men ikke strengt nødvendig.

Løsningsforslag

Kode i løsningsforslaget er basert på Python3, men må anses som en skisse og ikke som fullstendig, kjørbar kode. Skissen bør imidlertid være tilstrekkelig for å illustrere omfang og korrekthet på løsning.

Syntaks fra andre OO språk er også godtatt.

Del 1

Et universitet har behov for et system for å holde rede på studenters karakterbok. Så langt har universitetet benyttet et system skrevet i Python av tidligere studenter.

Dette systemet har blitt utvidet etterhvert som nye behov har meldt seg, men nå har det blitt tungvint å vedlikeholde og legge til nye funksjoner.

Universitetet ber deg lage en ny løsning som er objekt-orientert og som lar seg enhets-teste (unit-teste), og stiller følgende funksjonelle krav:

1. Hold rede på student, fag og karakter.
2. Systemet skal støtte at en karakterbok har flere studenter, en student tar flere fag, og hvert fag har flere karakterer.
3. Det skal kunne registreres karakter på student for hvert fag.
4. Det skal kunne beregnes gjennomsnittskarakter for hver student.
5. Karakterene i et fag skal ha vekting, slik at forskjellig vektede karakterer teller ulikt i gjennomsnittet.

Oppgave 1a - 15%

Implementér de klassene du anser som nødvendige. Du trenger ikke ta med andre metoder enn __init__().

Løsningsforslag

Her forventes det:

1. En definisjon på klassene og deres attributter.
2. Det er ikke nødvendig å implementere karakter som namedtuple, men det teller positivt.

```
import collections
Grade = collections.namedtuple('Grade', ('score', 'weight'))

class Subject():
    def __init__(self):
        self._grades = []

class Student():
    def __init__(self):
        self._subjects = {}

class Gradebook():
    def __init__(self):
        self._students = {}
```

Oppgave 1b - 15%

Implementer metodene `registrer_student(self, navn)`, `registrer_fag(self, navn)`, `registrer_karakter(self, score, vekting)` i de klassene du mener de hører hjemme.

Du trenger ikke ta med den koden du implementerte over, men indiker tydelig hvilken klasse metodene tilhører.

Løsningsforslag

```
class Subject:
...
    def registrer_karakter(self, score, weight):
        grade = Grade(score, weight)
        self._grades.append(grade)
```

```
class Student:
...
    def registrer_fag(self, name):
        if name not in self._subjects:
            self._subjects[name] = Subject()
        return self._subjects[name]

class Gradebook:
...
    def registrer_student(self, name):
        if name not in self._students:
            self._students[name] = Student()
        return self._students[name]
```

Oppgave 1c - 15%

Implementer metoden `gjennomsnittskarakter(self)` slik at du kan finne gjennomsnittskarakteren for en gitt student, eksempelvis slik:

```
bok = Karakterbok()

kari = bok.registrer_student('Kari Nordkvinne')
matte = kari.registrer_fag('Kalkulus')
matte.registrer_karakter(75, 0.20)

print(kari.gjennomsnittskarakter())
```

Løsningsforslag

Her må gjennomsnittskarakter opprettes både på student og på fag.

```
class Student:
...
    def average_grade(self):
        total, count = 0, 0
        for subject in self._subjects.values():
            total += subject.average_grade()
        count = len(self._subjects)

        return total/count

class Fag:
...
   def average_grade(self):
        total, total_weight = 0, 0
```

```
        for grade in self._grades:
            total += grade.score * grade.weight
            total_weight += grade.weight

        return total / total_weight
```

Oppgave 1d - 5%

På grunn av tidligere erfaringer med bugs i systemet, krever universitetet at utviklerne av karakterboka benytter enhetstesting (unit testing).

Forklar kort hva enhetstesting er, og implementer en (enkel) test for en av komponentene i systemet.

Løsningsforslag

Enhetstesting betyr at man utvikler tester som tester enkeltkomponenter (enheter) i komplekse systemer hver for seg. En test er vellykket når en komponent oppfører seg som definert i designet. Dette kan automatiseres, og når alle testene er vellykket er systemet godkjent. OO design og enhetstesting passer godt sammen fordi et godt OO design med veldefinerte komponenter og gjenbruk av kode egner seg veldig godt til enhetstesting.

Flere tester er mulige:

```
def test_registrer_student(student_name = None):
    book = Gradebook()
    student = book.student(student_name)
    if len(book._students) == 1:
        return True
    else:
        print("Length␣of␣student␣book␣should␣be␣1,␣but␣is",
            len(book._students))
```

Del 2

Oppgave 2 - 20%

Design patterns kan grovt inndeles i tre hovedkategorier:

1. Kreasjonelle ("Creational")
2. Strukturelle ("Structural")
3. Oppførselsbaserte ("Behavioral")

Forklar kort hva disse tre kategoriene representerer, og gi ETT eksempel fra hver kategori. Beskriv motivasjonen bak hvert eksempel du nevner, og hovedtrekkene i løsningen de skisserer.

Kode er ikke nødvendig med mindre du føler det gir en bedre beskrivelse.

Løsningsforslag

Her forventes det en klar beskrivelse av hver kategori, og hvert eksempel.

Oppgave 3 - 15%

```
class A:
    name = "Alfa"

    def __init__(self, foo):
        self.foo = foo
        foo = 100
        self.print_me()

    def print_me(self):
        print (self.name, self.foo)

class B(A):
    name = "Beta"
    def __init__(self, bar = 40):
        self.bar = bar
        print(self.name, bar)

class C:
    name = "Charlie"

class D(A, C):
    name = "Delta"
```

```
    def __init__(self, val):
        A.__init__(self, val)

    def print_me(self):
        print(self.name, "sier", self.foo)

a = A(20)
b = B()
d = D(60)
```

a) Angi hvilke attributter B arver fra A.

b) Forklar kort hva polymorfi er.

c) Angi hva programmet over skriver ut.

Løsningsforslag

---

a) 'name' arves men overskrives. 'print_me' arves og forblir uforandret. 'foo' arves ikke siden A sin init ikke kjøres.

b) Her forventes det en forståelse av skillet mellom grensesnitt og implementasjon. At flere ulike implementasjoner kan holde den samme "kontrakten", det være seg gjennom arv eller overstyring. I eksempelet over ser vi både arv og overstyring av foo i C.

c) "Alfa 20", "Beta 40", "Delta sier 60"

Oppgave 4 - 15%

Gi en beskrivelse av to (2) av de følgende uttrykk/konsepter:

a) Assosiasjon

b) Abstraksjon

c) Self (i Python)

d) Komposisjon

e) Arv

Løsningsforslag

Assosiasjon er et "har en" relasjon, hvor et objekt kan være komponert av andre objekter, men hvor disse objektene også kan eksistere på egenhånd.

Abstraksjon handler om å skjule implementasjonsdetaljer, gjerne gjennom et grensesnitt eller arv. Objekter kan brukes som en delt abstraksjon for kode og data. Abstraksjon tilrettelegger for gjenbruk av kode og data, og muliggjør at implementasjon kan endres uten å bryte kontrakter (grensesnitt brukt av andre klasser).

Self (i Python) er en referanse-syntaks som henviser til en spesifikk objekt-instans. Self er en del av namespace-syntaksen som lar koden skille mellom variabler på ulike nivå. Når man bruker self i koden henvises det til data som tilhører samme objekt som koden kjører i.

Komposisjon er et spesialtilfelle av assosiasjon, hvor de objektene et annet objekt består av ikke kan eksistere på egenhånd.

Arv mellom to klasser lar en subklasse arve attributter og metoder, og på den måten gjenbruke kode og data. Arv kan også gi polymorfi.
