IN1000: Obligatorisk innlevering 6

Frist for innlevering: 6.10.26, kl. 23:59 **Sist endret:** 28.9.26

Introduksjon

Det er kun quiz som *må* godkjennes denne uken, men erfaringsmessig er programmeringsoppgavene nedenfor svært viktige som grunnlag for oblig 7 (og eksamen!). Ved å levere løsningene dine får du tilbakemeldinger som hjelper deg videre.

Oversikt over undervisningen i IN1000 inkludert ekstratilbud finner du her.
Undervisningsmateriale for hver uke finner du her eller fra Timeplanen (under Forelesninger).

For å få godkjent obligatoriske innleveringer må du innen fristen:

1. Besvare denne quizen i Nettskjema. Det er ingen krav til antall rette.
2. I Devilry (se nederst):
    a. Levere skjermbilde (.jpg) av svarsiden i Nettskjema
    b. Levere filer som angitt i oppgaver merket **obligatorisk** nedenfor (0 eller flere). Merk at koden må kjøre uten feil, eventuelt må du beskrive feilen og hva du tror kan hjelpe (se under "Hvordan levere i Devilry")

Har innleveringen mindre feil eller mangler i obligatoriske oppgaver kan du få et (1) nytt forsøk - dette vil ha kort frist (3 virkedager) for innlevering, så sjekk e-post for tilbakemelding hver dag.

Læringsmål

Målet med denne oppgaven er å gi deg trening i å definere egne klasser og teste dem ved å opprette og bruke objekter av disse klassene. Videre skal du kjenne til sentrale begreper innen objektorientert programmering: Klasse, objekt, metode, konstruktør, instansvariabel og instansmetode, grensesnitt og innkapsling. Ta en titt på IN1000 ordlisten i tillegg til læringsmål og ressurser for uke 7.

Oppgave 1: Motorsykkel

Filnavn: *motorsykkel.py* og *test_motorsykkel.py*

I denne oppgaven gir vi en ganske detaljert fremgangsmåte med stegvise utvidelser og hyppig testing, som du bør bruke også i senere programmeringsoppgaver.
Du skal skrive en klasse, *Motorsykkel,* som skal modellere kjøring av motorsykler. En motorsykkel har et merke, et registreringsnummer og en kilometerstand som viser hvor langt den har kjørt.

1. Skriv klassen *Motorsykkel* med en konstruktør (*init*-metode) og de instansvariablene klassen trenger. Instansvariabelen for kilometerstand skal alltid ha startverdi 0. De to andre instansvariablene skal initialiseres med verdier fra init-metodens parametere.

2. Skriv en metode, *kjor(self, km),* som øker kilometerstanden med det gitte antall kilometer (*km*).

3. Skriv en metode, *hent_kilometerstand(self),* som returnerer motorsykkelens totale kilometerstand.

4. Du skal nå begynne på et testprogram for klassen *Motorsykkel*. Opprett en ny fil, *test_motorsykkel.py*. Øverst i filen skal du importere klassen *Motorsykkel*. Deretter skal du skrive en prosedyre *hovedprogram()* der det foreløpig bare står *pass* (dette erstatter du senere med innhold). Programmet skal kalle på denne. Kjør testprogrammet og rett opp eventuelle feil.

5. Inni *hovedprogram()* skal du nå opprette et objekt av klassen *Motorsykkel*, med et merke og et registreringsnummer som du velger selv. Opprett så to objekter til på samme måte. Kjør testprogrammet og rett opp eventuelle feil.

6. Utvid klassen *Motorsykkel* med en metode, *skriv_ut(self),* som skriver ut merke, registreringsnummer og kilometerstand på terminalen. Utvid deretter prosedyren *hovedprogram* i testprogrammet, med et kall på metoden *skriv_ut* på hvert av objektene du har laget. Kjør testprogrammet og rett opp eventuelle feil.

7. I testprogrammet skal du nå øke kilometerstanden på den motorsykkelen du laget sist med 10 km, og deretter skrive ut resultatet av et kall på *hent_kilometerstand* på terminalen. Kjør testprogrammet og rett opp eventuelle feil. Kontroller spesielt at kilometerstanden ble oppdatert riktig.

8. Frivillig/bonus: Legg til en *assert*-setning i prosedyren *hovedprogram* etter at du har oppdatert kilometerstanden (punkt 7), slik at du vil få en feilmelding (*AssertionError*) dersom resultatet av kallet på *hent_kilometerstand* ikke er som forventet.

   Tips: Metoder som returnerer en verdi kan kalles direkte i *assert*-setningen - for eksempel slik:
   <rett etter at et Motorsykkel-objekt er opprettet for variabelen mc>
   assert mc.hent_kilometerstand() == 0

Syntes du denne oppgaven var vanskelig? Se Trix-oppgave 7.01.
Syntes du denne oppgaven var enkel? Se Trix-oppgave 7.03.

Oppgave 2: Teorioppgave

Filnavn: *teori.txt*

Gi korte svar på spørsmålene under (hentet fra ukens forelesning):

1. Hva er innkapsling? Hvorfor er det nyttig?

2. Hva er grensesnittet til en klasse? Hvordan skiller det seg fra implementasjonen av en klasse?

3. Hva er en instansmetode (ofte kalt bare metode)? Hvordan skiller metoder seg fra prosedyrene/funksjonene vi har jobbet med hittil?

Oppgave 3: Hund

Filnavn: *hund.py* og *test_hund.py*

1. Skriv en klasse, *Hund*. Konstruktøren skal ha parametere som angir initialverdier (startverdier) til instansvariablene *alder* og *vekt*. *Hund* skal i tillegg ha en instansvariabel *metthet*, som alltid initialiseres med verdien 10.

2. Lag et testprogram i filen *test_hund.py,* slik at du kan teste klassen din etter hvert som du utvider den. Bruk designet fra oppgave 1 (legg alle testene i en prosedyre *hovedprogram*, som du må huske å kalle på). Foreløpig skal du kun opprette et objekt i testprogrammet, og deretter kjøre det.

   Endelig testprogram, slik det skal leveres, er beskrevet i punkt 6. Legg inn kall på hver metode du skriver og kjør testprogrammet underveis. Slik finner du feilene med en gang de oppstår. Om du ønsker kan du erstatte print-setningene i testprogrammet med assert-setninger.

3. Skriv metodene *hent_alder*() og *hent_vekt*() som henter ut (returnerer) henholdsvis *alder* og *vekt*.

4. Skriv en metode, *spring,* uten parametere (kun self). Metoden skal redusere *metthet* med 1. Dersom *metthet* blir mindre enn 5, skal metoden redusere *vekt* med 1.

5. Skriv en metode, *spis*. Metoden skal ha en parameter (et heltall) som angir mengde og som skal legges til *metthet*. Dersom *metthet* blir større enn 7, skal metoden også øke *vekt* med 1.

6. *hovedprogram*-prosedyren i *test_hund.py* skal opprette et hundeobjekt, kalle på metodene *spring* og *spis* minst 2 ganger hver samt skrive ut hundens *vekt* etter hvert metodekall. Avslutt programmet med en assert-setning som sjekker at endelig vekt er som du forventer.

   Tips: Her kan det være nyttig å bruke Pythontutor som debugger for å følge _metthet

og _vekt i Hund-objektet underveis. Kopier da inn linjene både fra klassedefinisjonen og testprogrammet i Pythontutor.

Syntes du denne oppgaven var vanskelig? Se Trix-oppgave 7.02, 7.05.
Syntes du denne oppgaven var enkel? Se Trix-oppgave 7.06.

Oppgave 4: Dato

Filnavn: *dato.py* og *test_dato.py*

1. Skriv en klasse, *Dato,* som representerer en dato. Konstruktøren til klassen skal ha parameterene *ny_dag*, *ny_måned* og *nytt_år*, og bruke disse til å initialisere instansvariabler i det nye objektet (*dag, måned* og *år*). Disse skal være heltall.
   Tips: Du kan bruke *int(s)* for å gjøre om en string *s* til et heltall - om *s* allerede var et heltall skjer det ingenting.

2. Begynn å skrive et testprogram i filen *test_dato.py,* på samme måte som i de tidligere oppgavene. Foreløpig trenger du bare å opprette ett *Dato*-objekt, med dag. måned og år som leses inn fra bruker i terminalen. For hver nye metode du skriver videre i klassen *Dato*, bør du teste at metoden virker som forventet. For å teste klassen med ulike datoer kan du kjøre programmet flere ganger.

3. Utvid klassen med metoder som utfører følgende operasjoner på en dato. Velg parametere til hver metode og om metoden skal returnere en verdi etter hva du synes er mest hensiktsmessig dersom det ikke er oppgitt. Begrunn gjerne valgene dine i kommentarer.
   a. En metode *hent_år*
   b. En metode som lager og returnerer en tekststreng som representerer datoen. Strengen skal være på en form som er grei å lese for mennesker, for eksempel “4.10.25”
   c. En metode (med en heltall-parameter *dag_nr)* som sjekker om dagen i objektets datoen er den samme som *dag_nr* og returnerer *True* eller *False*
   d. Frivillig/ bonus: En metode *ny_dag*, som endrer datoen i objektet til neste dag (kommenter eventuelle spesialtilfeller/ulike datoer metoden din ikke virker for)
   e. Frivillig/ bonus: En metode, *kommer_foer*, med parameter *annen_dato* (en streng på formen “6.10.25”) som sjekker om *annen_dato* er før dato-objektet metoden kalles på. Metoden skal returnere *True* (hvis datoen i parameteren er tidligere enn datoen i objektet) eller *False* (hvis datoen i parameteren er den samme eller senere enn datoen i objektet).

Testprogrammet ditt bør minimum inneholde det som står i punktene nedenfor. Bruk metodene i *Dato*-objektet du opprettet (gjør egne valg av verdier og annet ved behov).

- Skrive ut datoens år på terminalen
- Skrive ut “Lønningsdag!” på terminalen hvis dag er 15, “Ny måned, nye muligheter” om dag er 1.
- Skrive ut datoen på terminalen på en lesbar form

- (Hvis du har gjort 2d: Endre datoen i et *Dato*-objekt til dagen etter og skrive den ut på nytt)
- (Hvis du har gjort 2e: Les inn en ny dato fra bruker og skriv ut på terminalen om den nye datoen kommer før den datoen objektet ditt representerer)

Syntes du denne oppgaven var vanskelig? Se Trix-oppgave 7.02, 7.05.
Syntes du denne oppgaven var enkel? Se Trix-oppgave 7.04, 7.09.

Oppgave 5: Viktige begreper i objektorientert programmering

Filnavn: *begreper.txt*

Se illustrasjonseksempelet nedenfor om du er i tvil om hvordan oppgaven skal besvares.

For hvert av begrepene i listen under, oppgi et linjenummer i programmet der du finner et eksempel på begrepet. Hvert begrep er gjennomgått i uke 7. Hvis eksempelet går over flere linjer, oppgir du start- og sluttlinje.

A. Klassedefinisjon
B. Opprettelse av nytt objekt
C. Definisjon av instansvariabel
D. Definisjon av instansmetode
E. Definisjon av konstruktør
F. Kall på instansmetode
G. Argumenter

```python
class Menneske:
    def __init__(self, navn, hoeyde, vekt):
        self._navn = navn
        self._hoeyde = hoeyde
        self._vekt = vekt

    def spiser(self, mat, antKalorier):
        print(self._navn + " spiser " + mat + ".")
        self._vekt += antKalorier / 8000  #ca. antall kalorier i en kg kroppsfett

    def sover(self, timer):
        print(self._navn + " sover " + str(timer) + " timer.")
        self._vekt -= (timer * 75) / 8000 #ca 75 kalorier forbrent per timer

    def veie(self):
        print(self._navn + " veier " + str(self._vekt) + " kg.")

print()
per = Menneske("Per",175,80)
paal = Menneske("Paal",180,90)
per.veie()
paal.veie()

print()
per.spiser("ribbe",1000)
paal.spiser("surkaal",300)
per.veie()
paal.veie()

print()
per.sover(8)
paal.sover(7)
per.veie()
paal.veie()

```

Illustrasjonseksempel

Oppgave:

For hvert av begrepene i listen under, oppgi et linjenummer i programmet der du finner et eksempel på begrepet. Hvis eksempelet går over flere linjer, oppgir du start- og sluttlinje.

A. variabeldeklarasjon
B. utskriftsetning
C. innlesing fra terminal
D. beslutning med flere alternativer

```
1 tall = int(input("Oppgi et tall: "))
2 if tall < 10 :
3     print ("Tallet er mindre enn 10")
4 else :
5     print ("Tallet er 10 eller høyere")
6
7
```

Mulige riktige svar på spørsmålene:

A. linje 1
B. linje 3
C. linje 1
D. linje 2-5

Oppgave 6: Egen oppgave

1. Skriv oppgavetekst til en oppgave som handler om klasser og objekter! Eller du kan prøve følgende forslag:

   Skriv en klasse, *Person*, med en konstruktør som tar imot navn og alder og oppretter og initialiserer instansvariabler med disse. I tillegg skal konstruktøren opprette en instansvariabel, *hobbyer,* som en tom liste . Skriv en metode, *legg_til_hobby,* som tar i mot en tekststreng og legger den til i *hobbyer*-listen. Skriv også en metode, *skriv_hobbyer*. Denne metoden skal skrive ut alle hobbyene etter hverandre på en linje. Gi deretter Person-klassen en metode, *skriv_ut*, som i tillegg til å skrive ut navn og alder kaller på metoden *skriv_hobbyer*.

   Lag deretter et testprogram for klassen Person, der du lar brukeren skrive inn navn og alder, og oppretter et Person-objekt med informasjonen du får. Deretter skal brukeren, ved hjelp av en løkke, få legge til så mange hobbyer de vil. Når brukeren ikke lenger ønsker å oppgi hobbyer, skal informasjon om brukeren skrives ut.

2. Løs oppgaven! Du skal levere **både oppgaveteksten (enten forslaget gitt i blått ovenfor eller en du lager selv) og besvarelsen**. Skriv oppgaveteksten som kommentarer over løsningen din. Oppgaven skal være et eget arbeid, og skal skille seg fra tidligere egenoppgaver og andre obligatoriske oppgaver.

Krav til innlevering

- Programmer som leveres inn må kjøre uten feil. Om du har feil du ikke selv finner ut av kan du få hjelp i gruppetimer, på Fredagspython eller i Discourse. Hvis du leverer programmer med feil, må dette kommenteres i Devilry (se 4.c nedenfor).
- Koden din skal inneholde gode kommentarer som forklarer hva programmene gjør.
- Programmene skal ha gode utskriftsetninger som er enkle å forstå for bruker.

Hvordan levere i Devilry

1. Du skal IKKE lage en .zip-fil
2. Logg inn i Devilry.
3. Last opp alle filer som angitt i besvarte oppgaver.
4. Du **skal** også besvare noen spørsmål i kommentarfeltet i Devilry:
    a. Hvordan syntes du innleveringen var? Hva var enkelt og hva var vanskelig?
    b. Hvor lang tid (ca.) brukte du på innleveringen?
    c. Var det noen oppgaver du ikke fikk til? Hvis ja:
        i. For hvilke(n) oppgave(r) fungerer ikke det innleverte programmet ditt?
        ii. Hvorfor tror du programmet ikke fungerer?
        iii. Hva ville du gjort for å få programmet til å fungere hvis du hadde mer tid?
5. Husk å trykke **«Legg til innlevering eller spørsmål»** og sjekk deretter at innleveringen din er komplett. Du kan levere så mange ganger du vil frem til fristen, men **siste innlevering må inneholde alle filer som skal vurderes**.
6. Innleveringen med tilhørende Trix-oppgaver er minimum av hva du bør ha programmert i løpet av en uke - de fleste vil trenge en del mer trening for å få god kontroll på stoffet. Du finner flere oppgaver for denne uken her.

Finner du feil i oppgaveteksten, send en epost til siriamj@uio.no.