Objektorientert programmering i Python: Introduksjon

IN1000

Høst 2026 – uke 7

Siri Moe Jensen

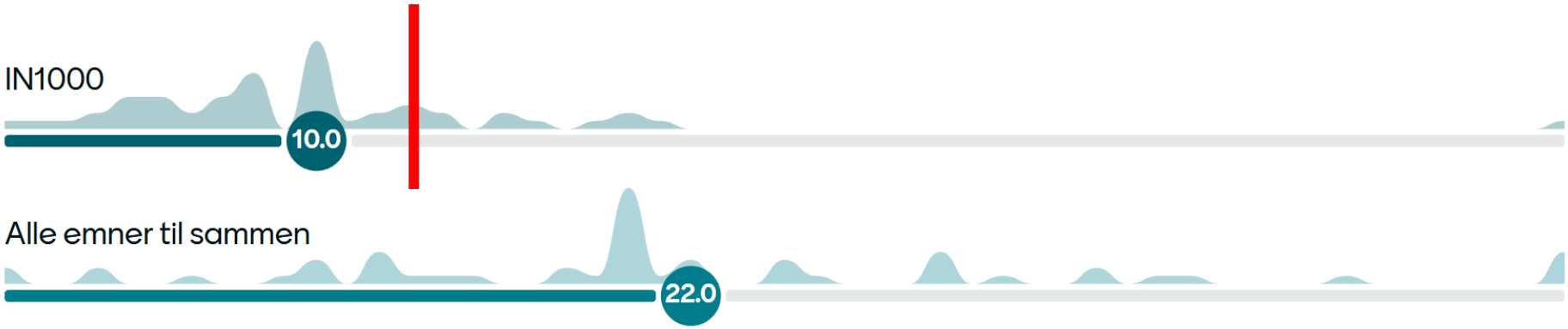
Innhold

- Underveisevalueringen og arbeid fremover
- Hvorfor objektorientert programmering?
- Programmering med objekter
- Hvordan definere og bruke egne klasser?

- Denne uken: Mye hvordan!
- Neste uke: Mer hvorfor?

Fra underveisevaluering

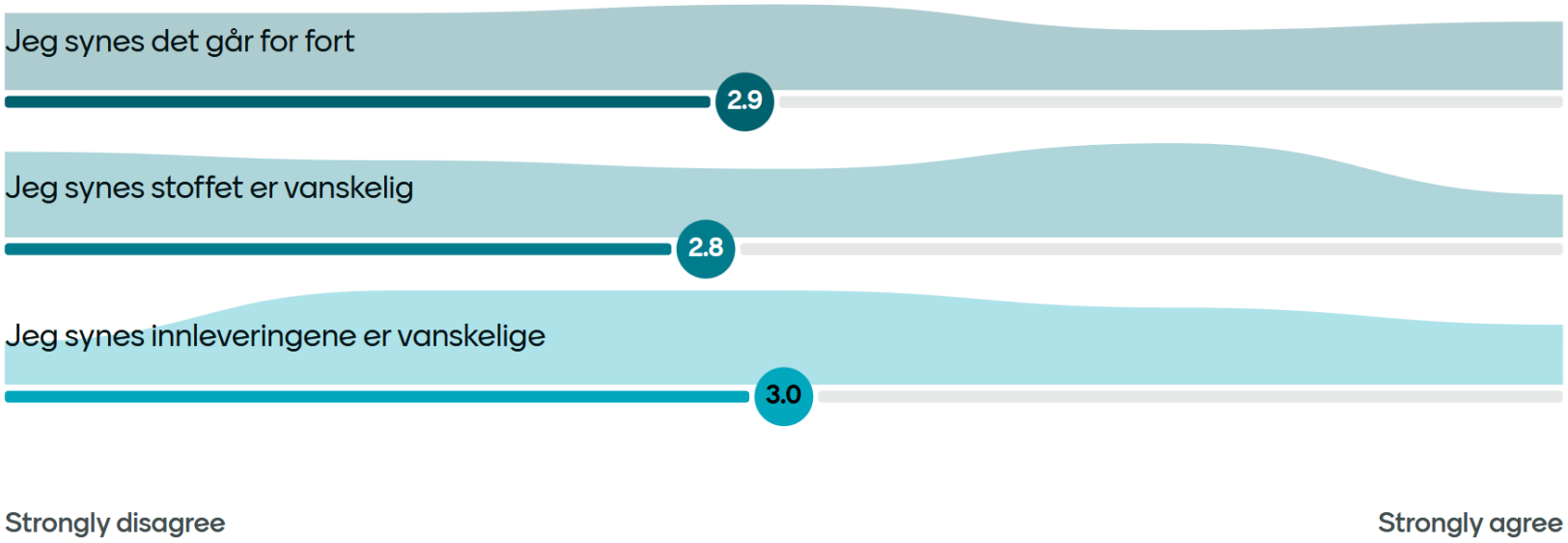
Hvordan bruker du tiden din på studiet (antall timer per uke)?



Fra underveisevaluering

Føler jeg klarer trix opg ganske greit men når det kommer til eksamen er det plutselig 10x vanskeligere

Lett eller vanskelig?



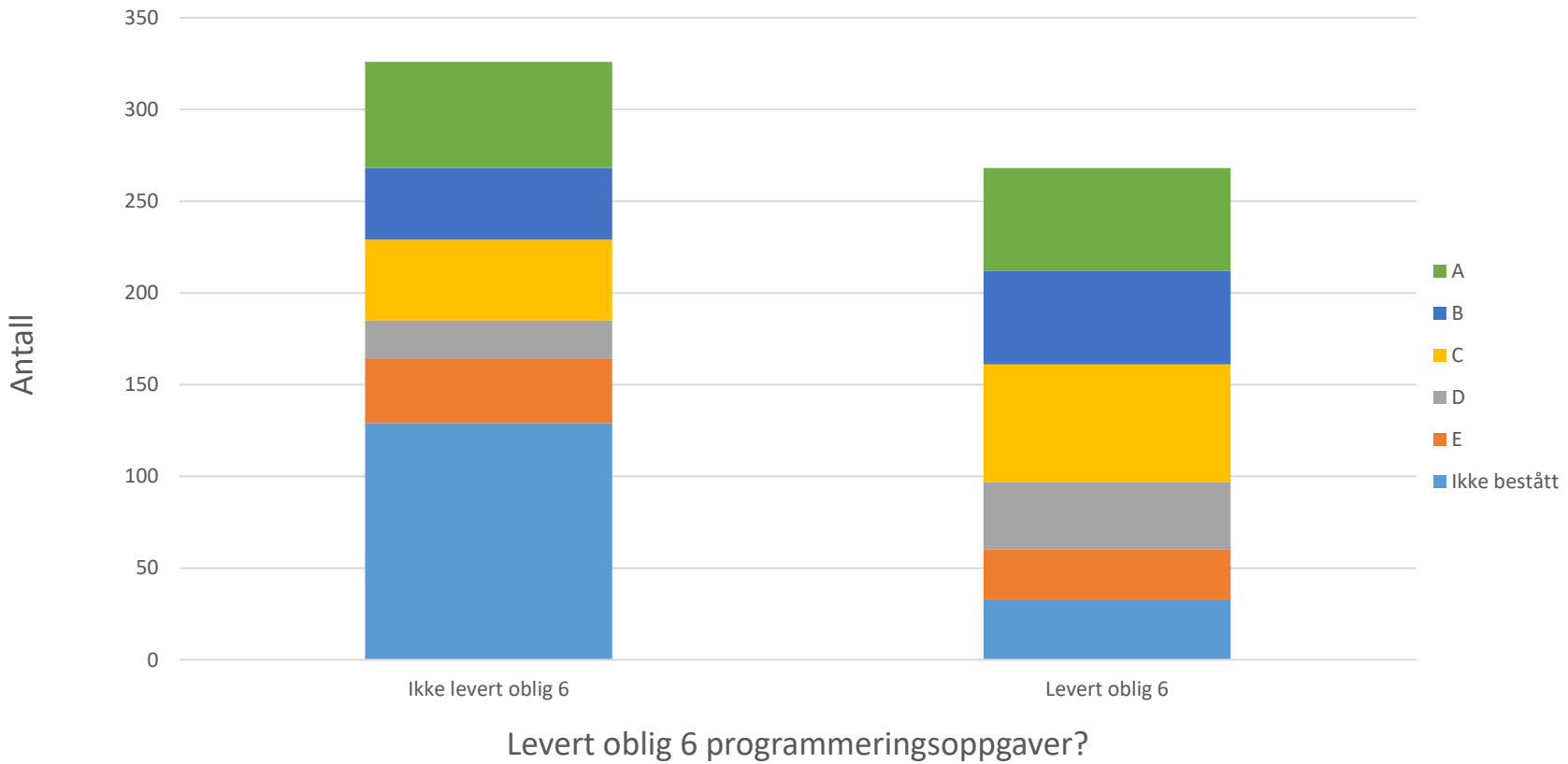
Annet fra underveisevaluering

- Noen ønsker/ spørsmål kan løses ved å gå gjennom (hele) semestersiden
- Man kan gå på flere og andre grupper enn man er satt opp på
- Fredagspython: Løs oppgaver med gruppelærer tilstede

Oblig 6 er ikke en pause mellom 5 og 7

- “Pausen” var forrige ukes forelesning (lite nytt programmeringsstoff)
- Hele oblig 6 (og gruppetimer og Trix-oppgaver) er helt nødvendig for å kunne løse oblig 7 selvstendig
- Jo lengre tid du trenger for å løse oblig 6, desto viktigere er det. Desverre.

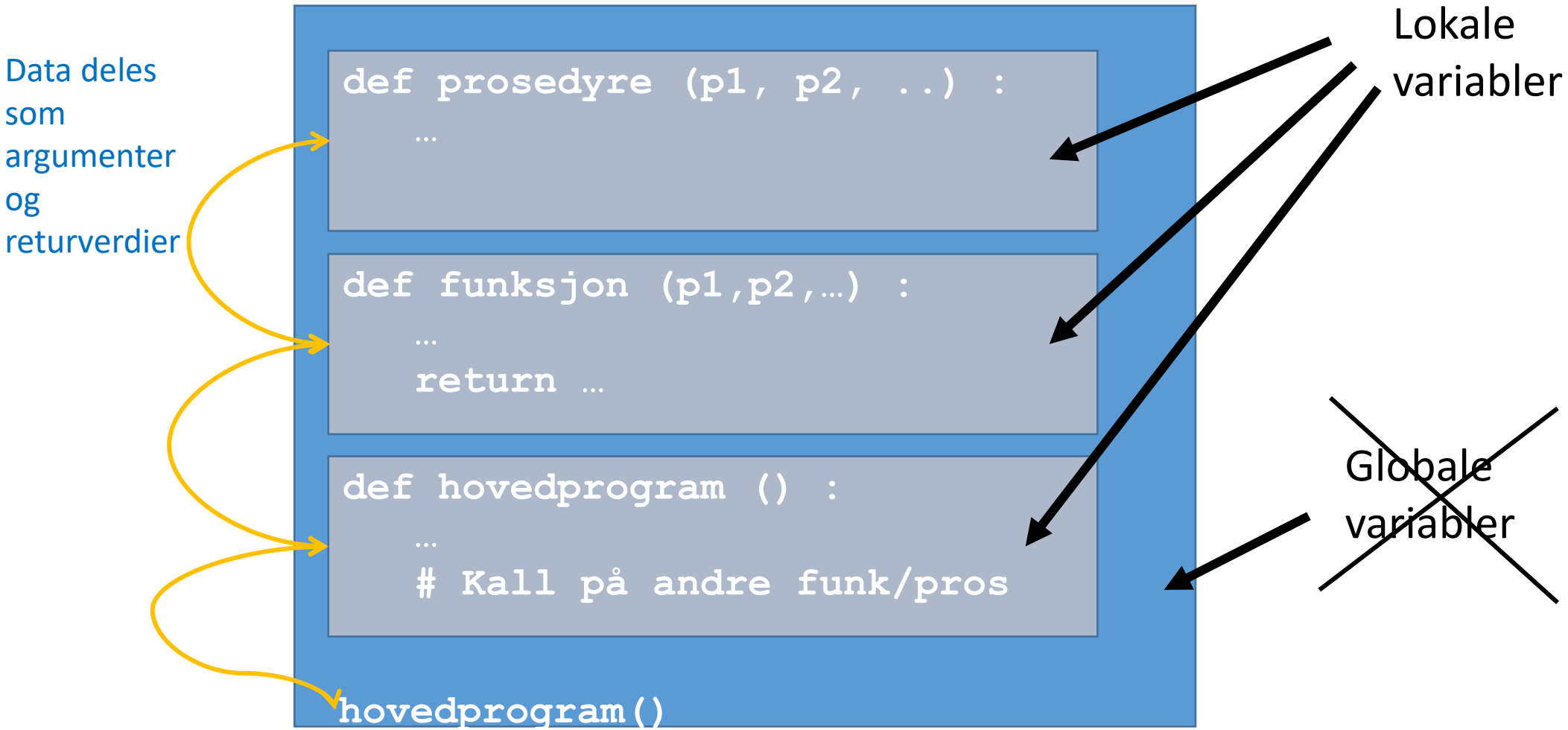
Korrelasjon oblig6 og eksamen (2024)



Modularisering
Splitt & hersk

Lite tilbakeblikk: Programflyt og skop

mittProgram.py



Hvorfor objektorientert programmering?

Bakgrunn

- 2. verdenskrig -> : Gradvis større og mer komplekse programmer i språk med (noen av) de mekanismene dere har lært hittil.
- Storforbruker: NACA/ NASA, under krigen og romfart-kappløpet
- Hovedsakelig matematiske beregninger – komplekse, men formelt definerte - store tallmengder, men repetitive operasjoner.
- Kompleksitet => dyrt, ufleksibelt og feilutsatt. Arbeidsdeling og gjenbruk ble vanskelig.

"computere" – menneskelige regnemaskiner (Hidden figures, 2016)



Økende maskinell behandling -> **kontrollregning** og **programmering**



Hva skjedde på 50- og 60-tallet?

- Datamaskinene ble kraftigere
- billigere
- mer utbredt

=> Nye anvendelser!

- Helsevesenet
- Trafikk-planlegging
- Bank, økonomisystemer

⇒ Nye tankesett, teori og programmeringsspråk for å håndtere størrelse og kompleksitet

Hvordan får man oversikt over og struktur på dette??

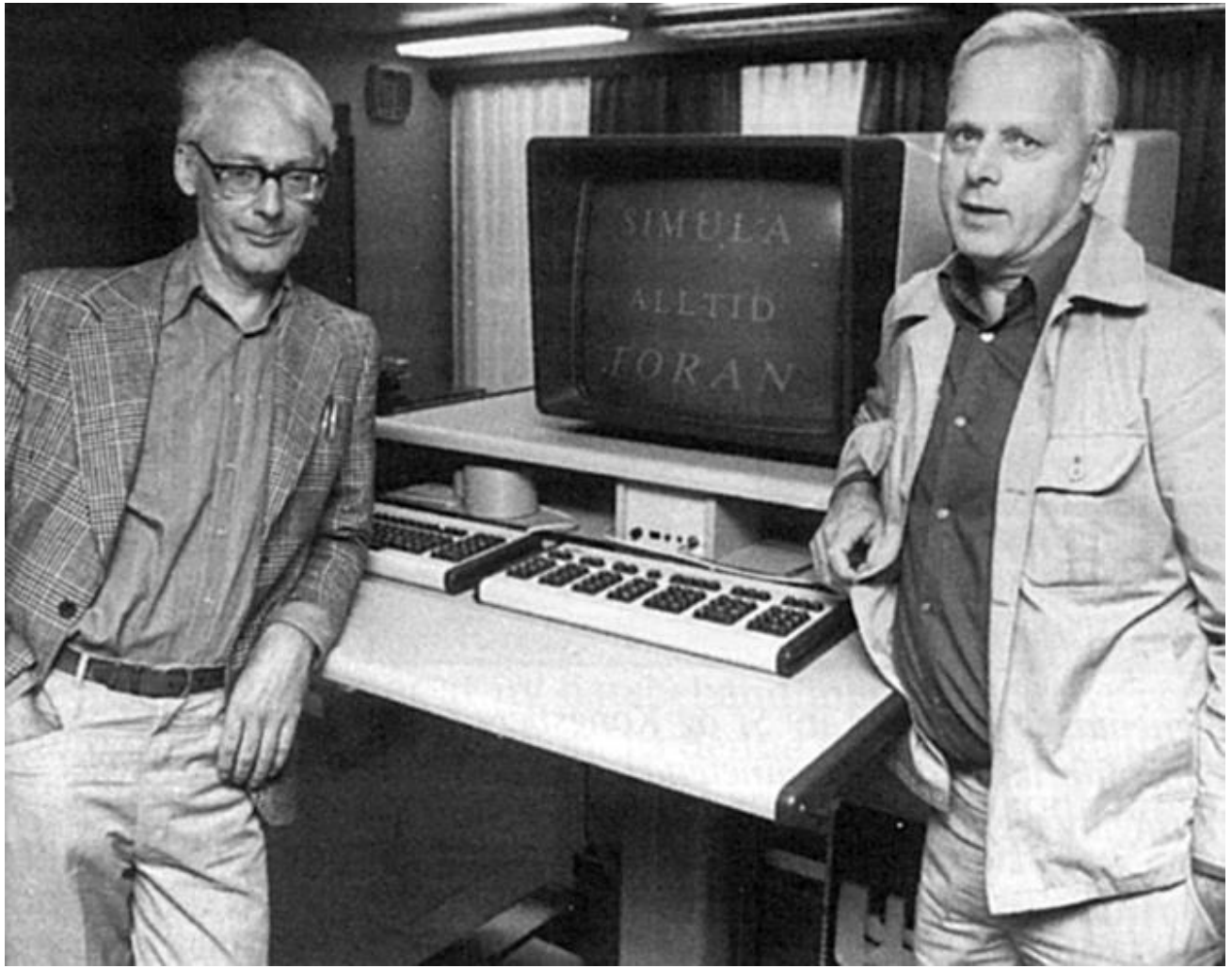




Hvorfor er de to informatikk-byggene oppkalt etter Kristen Nygård og Ole-Johan Dahl?

Tildelt Turing-prisen 2001 (informatikkens Nobelpris)

«.. *for ideas fundamental to the emergence of object oriented programming, through their design of the programming languages Simula I and Simula 67*»



se f. eks. Wikipedia "Structured programming" og "Programming paradigms"

Programmeringsspråk og -paradigmer

- Maskinnære språk (maskinkode, assemby-kode, LMC)
  - Kan hoppe til "vilkårlige" steder i programmet/ minnet
- Strukturert programmering ble et begrep på 60-tallet
  - erstatter JUMP/ GOTO med konstruksjoner som beslutninger, løkker, subrutiner
    - unntak/ diskusjonstema: Tidlig return, exceptions, break/ continue
  - støttes av *imperative språk* som tilbyr
    - Prosedural programmering (det dere har gjort hittil)
    - Objektorientert programmering (det vi starter med nå)
- Senere kom *deklarative språk*, f eks funksjonelle eller logiske språk

Objektorientert programmering: Kortere vei fra (kompleks) virkelighet til program

- Prosedyrer og funksjoner kan brukes for å løse deloppgaver

Modularisering
Splitt & hersk

- Hva om oppgavene handler om å bearbeide felles, kanskje komplekse, data?

- For eksempel trafikkregulering i et veikryss:
  - veier, veifelt, fotgjengeroverganger, trafikklys
  - endring og koordinering av lys basert på tidsintervaller, trykk på knapper, registrering av kjøretøy
  - Mange variabler, kompleks struktur, mange funksjoner => lett å miste oversikt

En måte å strukturere programmer på

- Hva om oppgavene handler om å bearbeide felles, kanskje komplekse, data?

Løsning:

- Samle relaterte data og kode for å manipulere dataene i **objekter**
- Objektene tilbyr resten av programmet et sett **metoder** grensesnitt
- Hvordan objektet representerer og manipulerer sine data er skjult innkapsling

Må fortsatt avgjøre: Hvilke data og operasjoner trengs? Hvordan gruppere i objekter?

Konseptet objektorientering (sitater fra Simula 50 år)

- Å programmere er å **modellere**
  - Vi bygger en egen representasjon av *fenomener* (konkrete eller abstrakte) vi trenger å manipulere i datamaskinen
  - denne representasjonen *er* ikke virkeligheten – men kan holde rede på noen aspekter som er viktige for det vi skal lage et program for

=> Vi lager en *modell av "virkeligheten"*

- Å programmere er å **forstå**
  - virkeligheten
  - det problemet/ behovet vi skal løse
  - hva som kan/ vil endre seg i fremtiden

Et objekt representerer "noe"

Fysisk, konkret eller abstrakt

- en bil
- en dato
- en student
- et informatikk-emne
- en sanginnspilling
- et byggeprosjekt

Eksempler i et grafisk brukergrensesnitt:

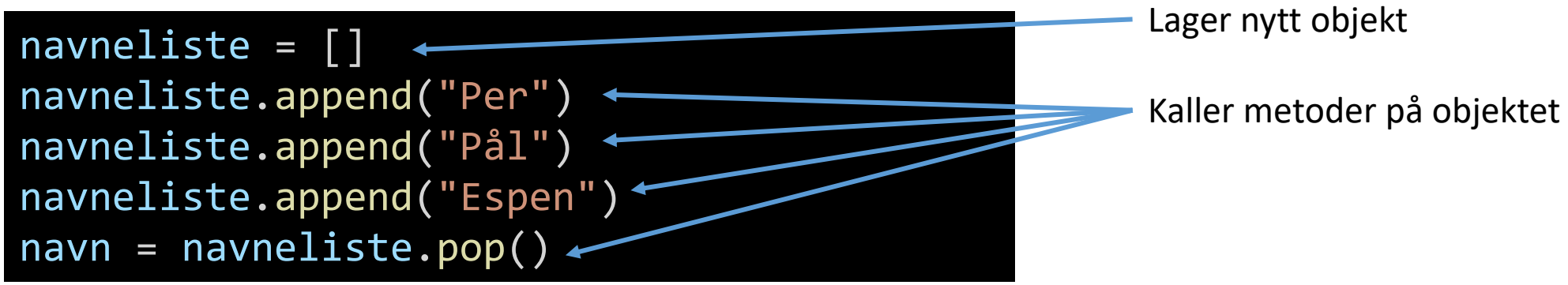
- et vindu
- en meny
- et felt i et vindu

Å programmere med objekter

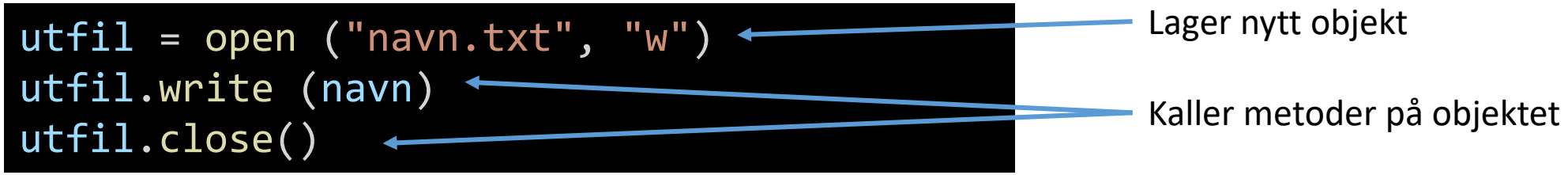
Objekter er *instanser* av en klasse

Vi har tidligere brukt objekter som

- ..representerer **lister** m/ tilhørende tjenester



- ..representerer **filer** m/ tilhørende tjenester



Klasser

- Objekter har ulik oppførsel – tilbyr ulike tjenester avhengig av hvilken *klasse* de tilhører

- Klassedefinisjonen bestemmer
  - hvilke tjenester objektene tilbyr i form av metoder
  - hvilke data objektet skal huske og hvordan de representeres

- Når vi har definert en klasse, kan vi opprette så mange objekter vi ønsker av denne i programmet vårt.

- Når programmet stopper, «dør» alle objektene

Metoder

- Vi kaller på metoder omtrent som prosedyrer og funksjoner
- MEN en metode må alltid kalles for ett bestemt objekt, derfor bruker vi "dot-notasjon"

```
liste = []
liste.append("ett_element")
```

- Vi utfører metoder på objektet vårt ved hjelp av kallet
  < objekt.metode(argumenter) >

- En metode kan ta argumenter (som en prosedyre/ funksjon)
- En metode kan returnere en verdi (som en funksjon)

Oppgave

- Finn et eller flere kodeeksempler der du har brukt "dot-notasjon" i et program.

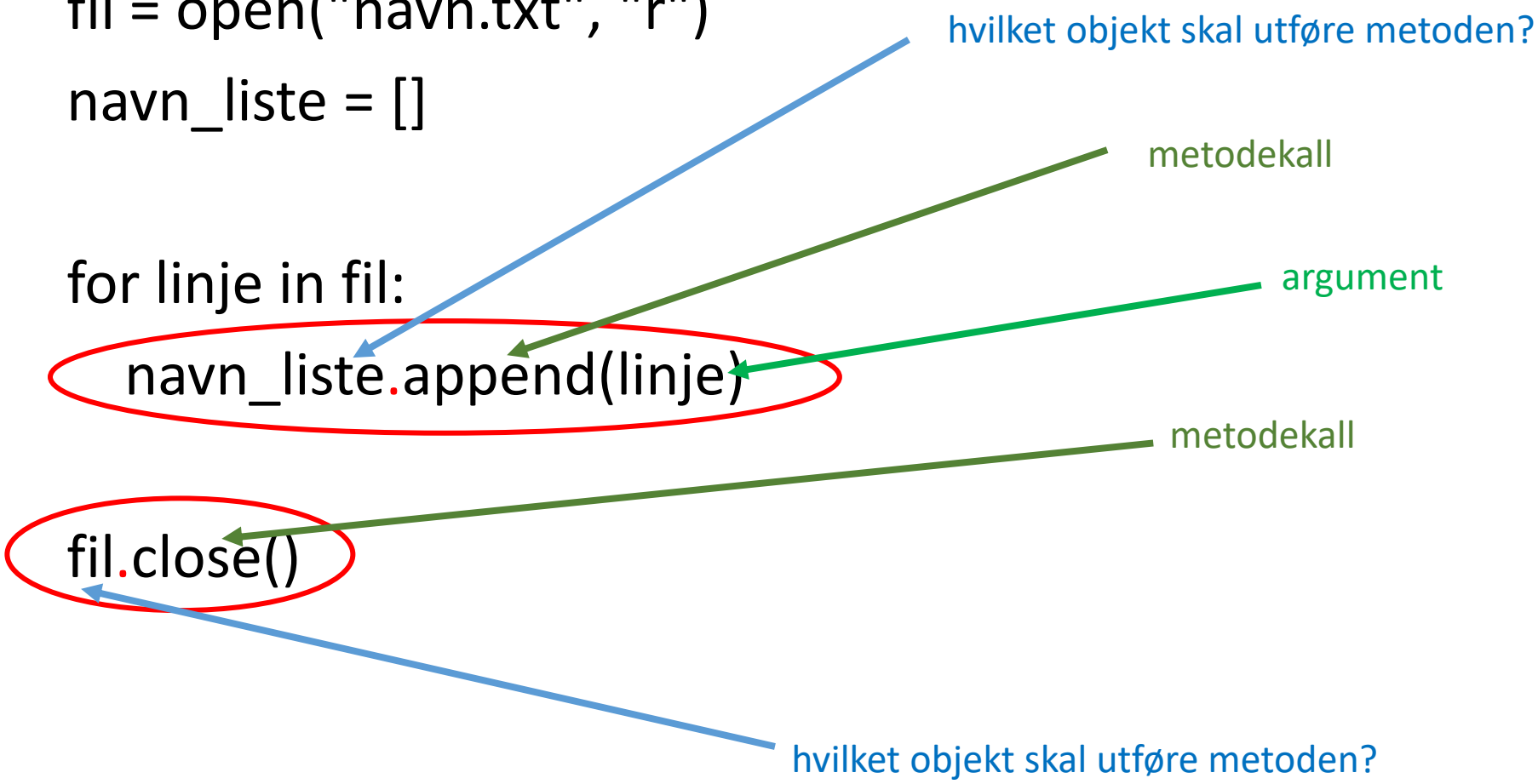
- For hvert eksempel: Identifiser
  - objektet
  - metodekall
  - argument(er)

Fra løsningsforslag Trix 5.01

```
fil = open("navn.txt", "r")
navn_liste = []

for linje in fil:
    navn_liste.append(linje)

fil.close()
```



Å lage egne klasser

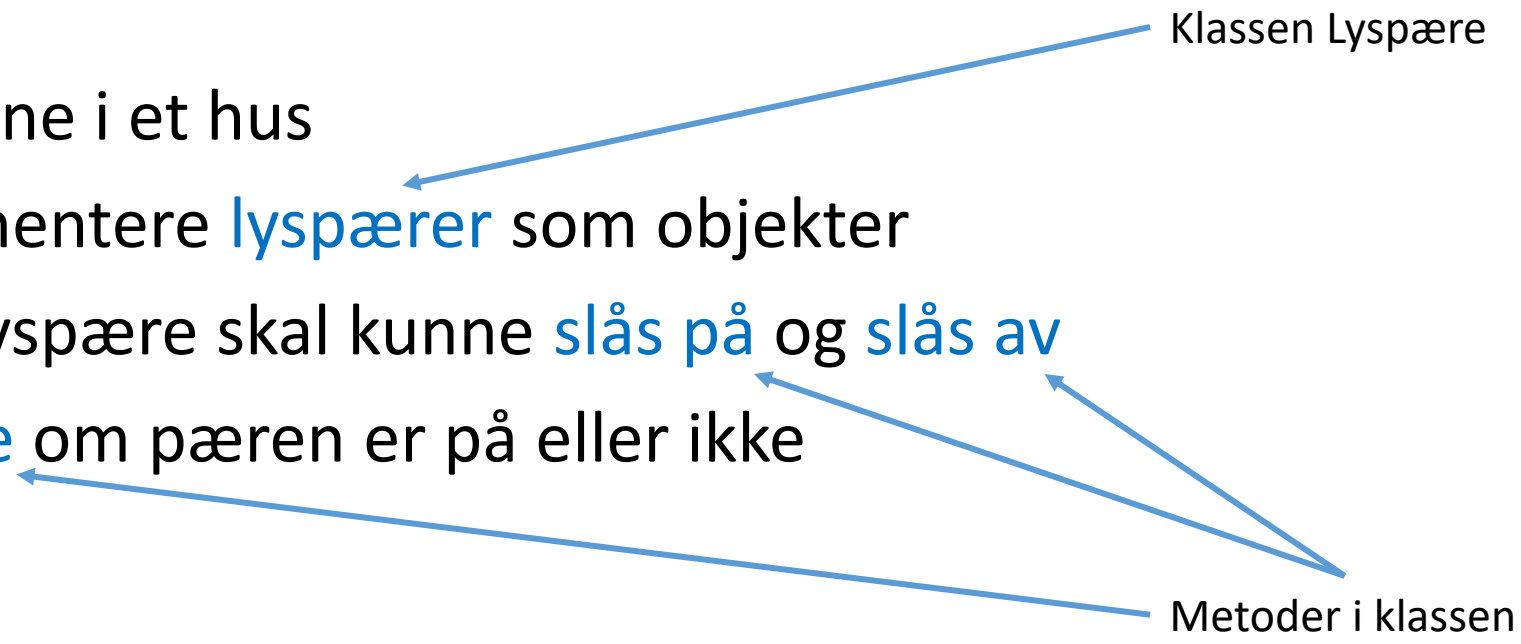
Hvordan kan vi lage objekter som tilbyr de tjenestene vi ønsker?

“Skreddersydde” objekter

- De innebygde klassene vi har brukt gir oss objekter som er nyttige i veldig mange programmer (strenger, lister, filer)
- Men objekter kan også løse mer spesialiserte oppgaver i ett program

Eksempel: Styre lysene i et hus

- Vi ønsker å implementere lyspærer som objekter
- Hver individuelle lyspære skal kunne slås på og slås av
- Vi må kunne sjekke om pæren er på eller ikke



Styring av smarte LED lyspærer

- Hva ønsker vi å kunne *gjøre* med lyspærene (versjon 0.1)?
  - Lage en ny lyspære (som starter avslått)
  - Sjekke om lyspæren er på `er_på`
  - Slå lyspæren på (tenne lyset) `tenne`
  - Slå lyspæren av (slukke lyset) `slukke`

Nå har vi (uformelt, med tekst) beskrevet hvilke *tjenester* lyspære-objekter skal tilby. Disse implementeres som *metoder* i en klasse Lyspære.

Hva må en lyspære holde rede på av data?

- Foreløpig: Kun om den er av eller på!
- Hvert Lyspære-objekt trenger en boolsk variabel, f eks

```
på
```

- Nå vet vi hvilke metoder Lyspære-objektene skal tilby, og hva slags data de trenger å holde styr på
- Da kan vi skrive en klasse for slike objekter!

Å lage (skrive, definere) en klasse

```
# Klasse for objekter som modellerer lyspærer
class Lyspære:

    # Konstruktør
    def __init__(self):
        pass

    # Er lyset på? Returnerer boolsk verdi
    def er_på(self):
        pass

    # Skrur lyset på (eller lar det være på)
    def tenne(self):
        pass

    # Skrur lyset av (eller lar det være av)
    def slukke(self):
        pass
```

{lyspære.py}

Ferdig klasse Lyspære

```
# Klasse for objekter som modellerer lyspærer
class Lyspære:

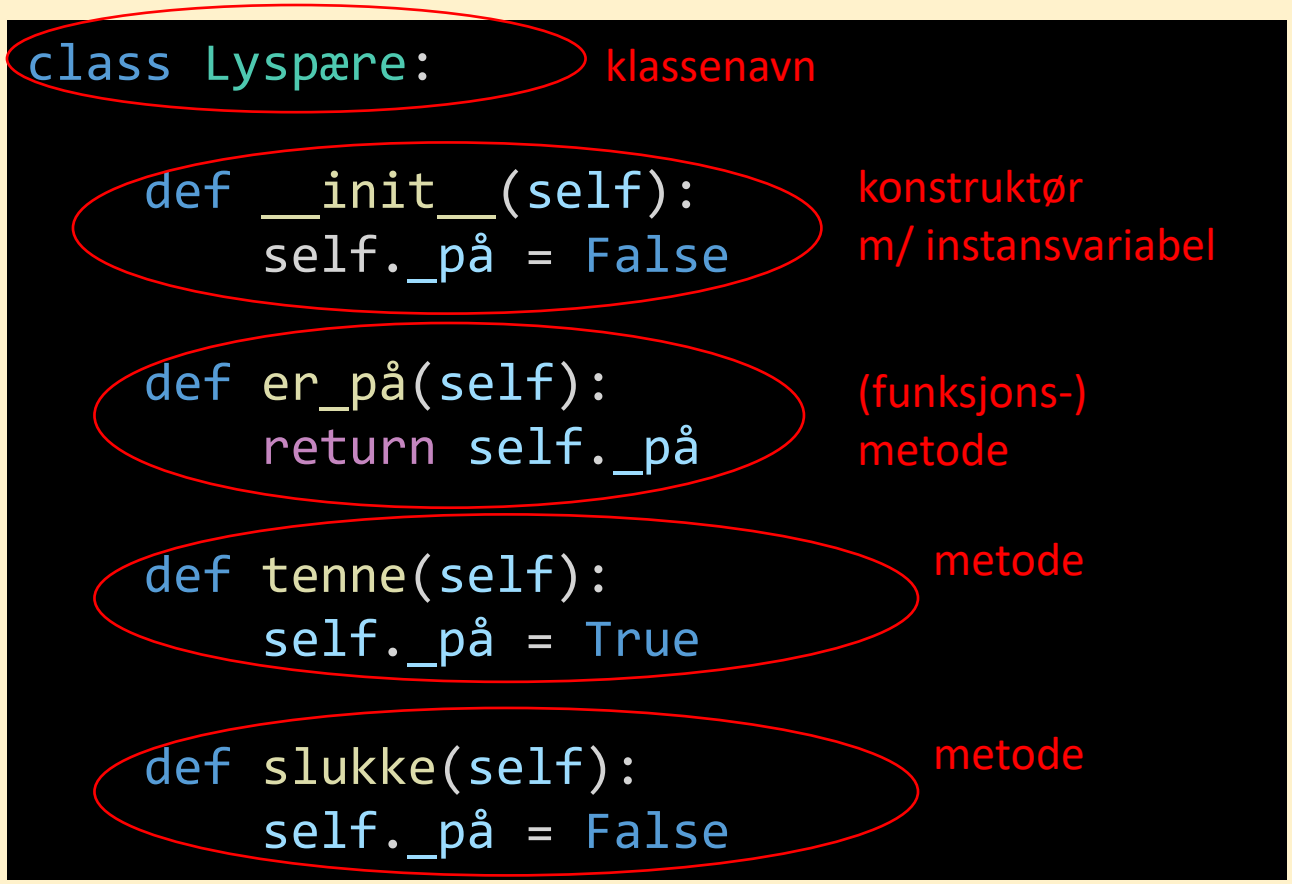
    # Konstruktør med instansvariabel
    def __init__(self):
        self._på = False

    # Er lyset på? Returnerer boolsk verdi
    def er_på(self):
        return self._på

    # Skrur lyset på (eller lar det være på)
    def tenne(self):
        self._på = True

    # Skrur lyset av (eller lar det være av)
    def slukke(self):
        self._på = False
```

Klasse-definisjon (syntaks)



Klasse-definisjon

```
class Lyspære:

    def __init__(self):
        self._på = False

    def er_på(self):
        return self._på

    def tenne(self):
        self._på = True

    def slukke(self):
        self._på = False
```

- Alle objekter av en klasse tilbyr samme metoder
- Hvert objekt av en klasse har (sine egne kopier av) samme instansvariabler
- Alle metodene i en klasse har tilgang til instansvariablene for et objekt

- Alle metoder må ha en parameter self, som husker hvilket objekt metoden ble kalt på
- Metodene bruker self for å aksessere instansvariablene i det objektet de er kalt for

Opprette objekter (syntaks)

- Noen innebygde klasser i Python har snarveier for opprettelse av objekter
  - "" oppretter (og returnerer) en ny **`str`**
  - [] oppretter (og returnerer) en ny **`list`**

```
min_streng = ""

min_liste = []
```

- For de klassene vi selv definerer, opprettes (og returneres) objekter vha *klassenavn med argumentliste* (som kan være tom):

```
utelys = Lyspære()
```

Kall på metoder i et Lyspære-objekt (syntaks)

- Et objekt tilbyr bare de metodene som er definert i klassen

```
utelys = Lyspære()

status = utelys.er_på()

utelys.tenne()
```

Kaller på metoden **er_på()** som returnerer en boolsk verdi

Kaller på metoden **tenne()** som endrer verdien i instansvariabelen **_på** til **True**

Et testprogram for Lyspære

```
class Lyspære:

    def __init__(self):
        self._på = False

    def er_på(self):
        return self._på

    def tenne(self):
        self._på = True

    def slukke(self):
        self._på = False
```

```
def hovedprogram():
    utelys = Lyspære()
    assert not utelys.er_på()

    utelys.tenne()
    assert utelys.er_på()

    utelys.slukke()
    assert not utelys.er_på()

    print("Alle tester gjennomført!")

hovedprogram()
```

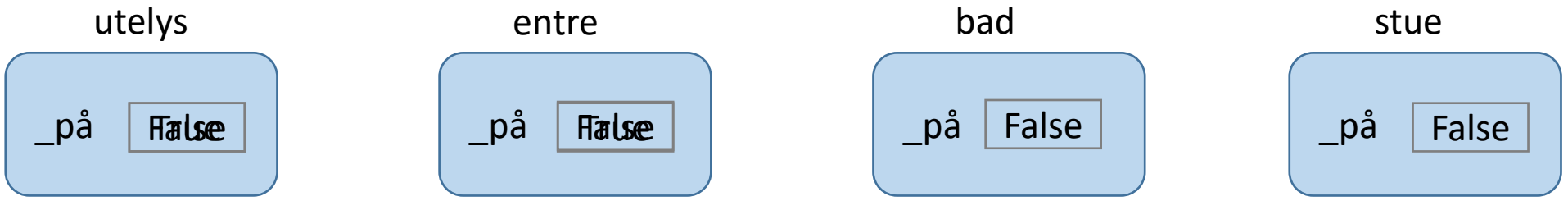
Instansvariabler og instansmetoder

- Hvert objekt som opprettes av en klasse får sine egne kopier av instansvariablene.
- Metodene har tilgang til instansvariablene i det objektet de kalles på

- Alle metodene i en klasse har tilgang til instansvariablene
- self-parameteren forteller metoden hvilket objekt den er kalt på
- alle instansvariabler må derfor aksesseseres med self i metodene

```
utelys = Lyspære()
entre = Lyspære()
bad = Lyspære()
stue = Lyspære()

utelys.tenne()
entre.tenne()
```



En ny klasse: Student

Når kan man typisk ha nytte av klasser?

- For å tilby et sett tjenester som hører sammen (verktøykasse)
- Spesielt nyttig for tjenester som trenger å dele data over tid (eks for håndtering av en liste, eller en fil, eller en tekst)
- For å kunne lage og bruke flere like objekter med samme type data og oppførsel (eks representere lyspærer eller studenter)
- For å simulere sammensatte fenomener vha objekter som representerer virkelige elementer (eks trafikk- simuleringer)

Fremgangsmåte

- Identifiser aktuelle klasser (hva er sentrale «ting»/ begreper programmet skal behandle?)
- Hva skal objekter av klassen tilby? (tjenester/ metoder)
- Hva skal objektene huske av data, hvilke variabler trenger vi?
- Skriv klassen og et enkelt testprogram



Eksempel Student

1. Hva skal objektet kunne gjøre?
2. Hvilke data må det huske?
3. Skriv klassen og lite testprogram

- Skal lagre og bearbeide informasjon om studenter

- Studenter har MANGE egenskaper – ulike systemer vil "se" studenter på ulike måter
  - Devilry
  - Eksamenssystemet
  - Lånekassen

  Vi må lage en *modell* av en student som (kun) omfatter *de egenskapene vi trenger*

- Hva skal jeg bruke klassen til?
  - Registrere poeng på studentens IN1000 innleveringer

Student

1. Hva skal objektet kunne gjøre?
2. Hvilke data må det huske?
3. Skriv klassen og lite testprogram

- Ønsker å kunne gjøre følgende:
  - Lage nye Student-objekter med 0 poeng ved semesterstart
  - Øke en students poeng (hvor mye kan variere) når ny innlevering er rettet
  - Lese av (hente ut) antall poeng for en student

- Dvs: Klassen må (i tillegg til konstruktør) ha metoder for:
  - Registrerer nye poeng:
    `registrer_poeng(self, antall_poeng)` `# endrer instansvariabel, må vite hvor mange poeng`
  - Henter antall poeng:
    `hent_poeng(self)` `# Returnerer antall poeng studenten har`

Hvilke data må et objekt huske?

1. Hva skal objektet kunne gjøre?
2. Hvilke data må det huske?
3. Skriv klassen og lite testprogram

- Antall poeng

  => vi trenger en instansvariabel som lagrer et heltall

Student – grensesnitt i Python

Oppgave: Fyll ut!

1. Hva skal objektet kunne gjøre?
2. Hvilke data må det huske?
3. Skriv klassen og lite testprogram

```
class Student:
    # Konstruktør, kun self-parameter
    # instansvariabel for poeng (int)
    def __init__(self):

    # Returnerer antall poeng
    def hent_poeng(self):

    # Øker poeng med antall
    def registrer_poeng(self, antall):
```

Student – implementasjon

1. Hva skal objektet kunne gjøre?
2. Hvilke data må det huske?
3. Skriv klassen og lite testprogram

```
class Student:
    # Konstruktør, kun self-parameter
    # instansvariabel for poeng (int)
    def __init__(self):
        self._poeng = 0

    # Returnerer antall poeng
    def hent_poeng(self):
        return self._poeng

    # Øker poeng med antall
    def registrer_poeng(self, antall):
        self._poeng += antall
```

Grensesnitt og innkapsling

- Grensesnitt er et sentralt begrep i programmering
  - Så lenge grensesnittet er stabilt (eller utvides), fungerer programmer som bruker det
  - Gjennomført ansvarsdeling (modularisering, splitt og hersk)
  - Spesielt nyttig når kode (klasser) skal brukes i flere/ store programmer/ av flere programmerere
- Metodene utgjør klassens (offentlige) grensesnitt
- Innkapsling innebærer at kun metoder kan brukes utenfor klassen
- Data (instansvariabler) i klassen er "non-public" og brukes kun inne i klassen

Innkapsling i Python og IN1000

- Python støtter innkapsling med *konvensjoner*
  - Alle navn som ikke synes i klassens grensesnitt får navn som begynner med _
  - Gjelder instansvariabler og eventuelle hjelpemetoder
- Dette prinsippet skal dere bruke konsekvent i IN1000

- I praktisk bruk er det et *designvalg* når og hvordan man anvender innkapsling
- Vi gjør valget enkelt og trener samtidig opp bevisstheten om grensesnitt
- Dere vil se eksempler på direkte aksess av data i IN1010

Forbedring av studentklassen

- Klassen Student gir oss mulighet til å opprette så mange student-objekter vi vil
- Objektene vil ha hver sin instansvariabel som lagrer antall poeng for én student
- Men hvordan vet vi hvilket objekt som lagrer poengene til hvilken student?

1. Vi kan ha en variabel som forteller oss hvilken student hvert objekt tilhører

```
ole = Student()
dole = Student() # Upraktisk å måtte skrive hvert navn inn i programmet
```

2. Vi kan ha en ordbok som kobler navn med objekt

```
studenter = {}
studenter["ole"] = Student()
studenter["dole"] = Student() # bedre, vi kan lage så mange vi vil
```

3. **Vi kan lagre studentenes navn i objektet**

Utvidelse av klassen Student

Vi ønsker å utvide klassen med en instansvariabel for studentens navn (i tillegg til studentens poengsum)

- Vi kan fortsatt lagre objektene i en ordbok med navn som nøkkel (mer om dette neste gang)
- Når vi oppretter et nytt objekt er det fordi vi har informasjon om en ny student – da kan vi oppgi navnet med en gang, det vil si til konstruktøren
- Konstruktøren trenger dermed en parameter `navn`
- Da kan man gi initialverdi for `navn` ved opprettelse av hvert objekt

Oppgave:
Ny klasse StudentMedNavn

```
class Student:
    def __init__(self):
        self._poeng = 0

    def hent_poeng(self):
        return self._poeng

    def registrer_poeng(self, antall):
        self._poeng += antall
```

Sammenlignet med Student-klassen til venstre skal du:

1. Endre konstruktør:
   - Ny parameter navn
   - Ny instansvariabel _navn

2. Ny metode for å skrive ut navn og poeng på terminal
   - ny metode skriv_ut

3. Hvordan opprettes et objekt av klassen StudentMedNavn?

Oppgave - løsningsforslag

```
class StudentMedNavn:
    def __init__(
        self._poeng = 0


    def hent_poeng(self):
        return self._poeng

    def registrer_poeng(self, antall):
        self._poeng += antall

    def skriv_ut(
```

Sammenlignet med Student-klassen til venstre skal du:

1. Endre konstruktør:
   - Ny parameter navn
   - Ny instansvariabel _navn
2. Ny metode for å skrive ut navn og poeng på terminal
   - ny metode skriv_ut
3. Hvordan opprettes et objekt av klassen StudentMedNavn?

Oppgave - løsningsforslag

```
class StudentMedNavn:
    def __init__(self, navn):
        self._poeng = 0
        self._navn = navn

    def hent_poeng(self):
        return self._poeng

    def registrer_poeng(self, antall):
        self._poeng += antall

    def skriv_ut(self):
        print(self._navn, "har totalt", self._poeng, "poeng")
```

Sammenlignet med Student-klassen til venstre skal du:

1. Endre konstruktør:
   - Ny parameter navn
   - Ny instansvariabel _navn
2. Ny metode for å skrive ut navn og poeng på terminal
   - ny metode skriv_ut
3. Hvordan opprettes et objekt av klassen StudentMedNavn?

Test av klassen

- Vi skriver et testprogram – foreløpig i samme fil
- Oppretter (minst) et objekt
- Kaller på alle metoder (minst) en gang - ulike sekvenser kan være nødvendig for å avdekke feil siden objekter husker verdiene sine (har en tilstand) mellom kall
- **`assert`** tvinger oss til å tenke ut på forhånd hva vi forventer - og forstyrrer ikke kjøringen når alt er ok

=> NB: Skriv testprogrammet tidlig, og test en og en metode (inkludert konstruktør)

Testprogram for StudentMedNavn-klassen

```
# <Klassedefinisjon>

def hovedprogram():
    # tester klassen Student_med_navn
    stud = StudentMedNavn("Einstein")

    assert stud.hent_poeng() == 0   # Ble objektet opprettet med forventet poengsum?
    stud.skriv_ut()                 # Er innholdet som forventet og fungerer skriv_ut?

    stud.registrer_poeng(5)
    assert stud.hent_poeng() == 5   # Ble antall poeng oppdatert som forventet?
    stud.skriv_ut()                 # Er innholdet fortsatt som forventet i skriv_ut?

hovedprogram()
```

Samlinger av objekter

- Av en klasse kan vi lage mange objekter med ulike verdier i instansvariablene (ulike tilstander)
- Dette er ofte veldig nyttig, og vi skal bruke det mye i IN1000
- Objekter kan lagres både i lister, ordbøker og mengder, og vi kan iterere over disse med for-løkker som vi har sett før
- En åpenbar fordel: Vi slipper å lage en egen variabel for hvert objekt

Organisering av kode

- I innleveringene fremover skal dere levere hver klasse i en egen fil (en modul).

- Testprogrammene importerer så denne modulen, og bruker klassen som om den lå i samme fil.

test_flere_studenter.py

Lite eksempel: liste med flere objekter

```python
from studentmednavn import StudentMedNavn

studenter_liste = []
studenter_liste.append(StudentMedNavn("Ole"))
studenter_liste.append(StudentMedNavn("Dole"))
studenter_liste.append(StudentMedNavn("Doffen"))

studenter_liste[0].registrer_poeng(3)
studenter_liste[0].registrer_poeng(5)
studenter_liste[0].registrer_poeng(5)
studenter_liste[0].registrer_poeng(6)

print("Studenter med nok poeng (fra liste):")
for student in studenter_liste:
    if student.hent_poeng() >= 19:
        student.skriv_ut()
```

Utskrift:

```
Studenter med nok poeng (fra liste):
Ole har totalt 19 poeng
```

Oppsummering

(se også notatet lenket fra uke 7-siden for 2 sider tekst)

Læringsmål uke 7

- Kjenne til motivasjon og bakgrunn for objektorientert programmering
- Programmering: Kunne definere en klasse, opprette objekter og kalle metoder på objektene
- Kunne bruke og forklare begrepene klasse, objekt, klassedefinisjon, konstruktør, metode og instansvariabel
- Forstå sentrale begreper som grensesnitt og innkapsling
- Kjenne til utviklingsprosessen for en klasse gjennom design, implementasjon og testing

OO støtter modellering og modularisering

- Både funksjoner og klasser er mekanismer som hjelper oss å strukturere koden vår gjennom modularisering

- Klassen definerer *data og operasjoner* som hører sammen, og kan definere et begrenset "vindu" (*grensesnitt*) som brukere av klassen (programmerere) får til objektene

- Klassedefinisjonen er mønsteret som blir fulgt når vi lager nye objekter av klassen: Alle objektene har de samme instansvariablene, men instansvariablene kan ha ulike verdier i hvert objekt

- Et objekt husker data i instansvariablene sine mellom hver gang en metode kalles for objektet

Å definere en klasse, opprette objekter og kalle metoder

```
class Lyspære:

    def __init__(self):
        self._på = False

    def er_på(self):
        return self._på

    def tenne(self):
        self._på = True

    def slukke(self):
        self._på = False
```

klassenavn

konstruktør m/ instansvariabel

(funksjons-) metode

metode

metode

```
# Testprogram for Lyspære

utelys = Lyspære()
assert not utelys.er_på()

utelys.tenne()
assert utelys.er_på()

utelys.slukke()
assert not utelys.er_på()

print("Alle tester gjennomført!")
```

Neste uke: Litt mer kjøtt på bena

- Hva er det egentlig som skjer når vi definerer klasser, oppretter objekter og kaller på metoder?

- `self` – og andre referanser

- Visualisering av objekter og datastrukturer

Det er vanskelig å forstå det man ikke husker, og å huske det man ikke forstår

=> Mengdetrening denne uken!