Objektorientert programmering i Python

IN1000

Høst 2026 – uke 8

Siri Moe Jensen

Læringsmål forrige uke (7)

Har fått et språk og blitt kjent med en syntaks for programmering med klasser og egne objekter

- 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

Plan for i dag

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

- Forrige ukes pensum: Flere eksempler, mer detaljer, mer forklaring

- Nytt: Referanser, immutable, None

- Datastrukturer
  - Klasser som bruker andre klasser
  - Organisering av mange objekter
  - Visualisering

Neste uke:

- Mer om datastrukturer med flere klasser og med objekter som refererer til andre objekter
- «Magiske» / spesielle metoder

Program med klasser

klasse.py

```
class Klasse:
    def __init__(self,p1):
        self._instvar1 = p1
        self._instVar2 = 3.14

    def metode1 (self, p2,..):
        …


def hovedprogram ():
    …
    # oppretter og bruker
objekter

hovedprogram()
```

Fungerer og kan brukes underveis, i små tester, og i Python Tutor.

Klasser kan være nyttige å gjenbruke i flere programmer

- For at det skal fungere mest mulig feilfritt og forutsigbart, må noen krav være oppfylt:

    1. Klassen må ha et stabilt og veldokumentert **grensesnitt**: Vi må vite hvilke metoder som tilbys, hva de gjør, hvilke argumenter de tar og om de returnerer en verdi
    2. **Innkapsling**: Instansvariabler i en klasse aksesseres kun fra metoder i klassen
    3. Klassedefinisjonen ligger i **en egen fil** (som kan importeres der den trengs). Unngå tester i samme fil før du leverer inn eller bruker klassen i andre programmer

Dette vil du bruke i oblig 7

Program med klasse i egen programfil

klasse.py

```
class Klasse:
    def __init__(self,p1):
        self._instvar1 = p1
        self._instVar2 = 3.14

    def metode1 (self, p2,..):
        …
```

program.py

```
from klasse import Klasse
def hovedprogram ():
    …
    # oppretter og bruker objekter
hovedprogram()
```

Klasser skrives ofte slik at de er nyttige i mange ulike programmer og kan gjenbrukes.

Eks: strenger og lister

Dette er slik dere leverer obliger (alle klasser i egne filer)

Visualisering: Struktur for program*teksten*

- Tegningene av hvordan vi kan strukturere programmene våre ved hjelp av funksjoner og klasser, i en eller flere filer, visualiserer **kildekoden (programmet)** vårt.

klasse.py

```
class Klasse:
    def __init__(self,p1):
        self._instvar1 = p1
        self._instVar2 = 3.14

    def metode1 (self, p2,..):
        …
```

program.py

```
from klasse import Klasse

def hovedprogram ():
    …
    # oppretter og bruker objekter

hovedprogram()
```

Programtekst versus programutføring

- VSCode og andre editorer viser oss programteksten/ kildekode, men ikke hva som skjer under utføring

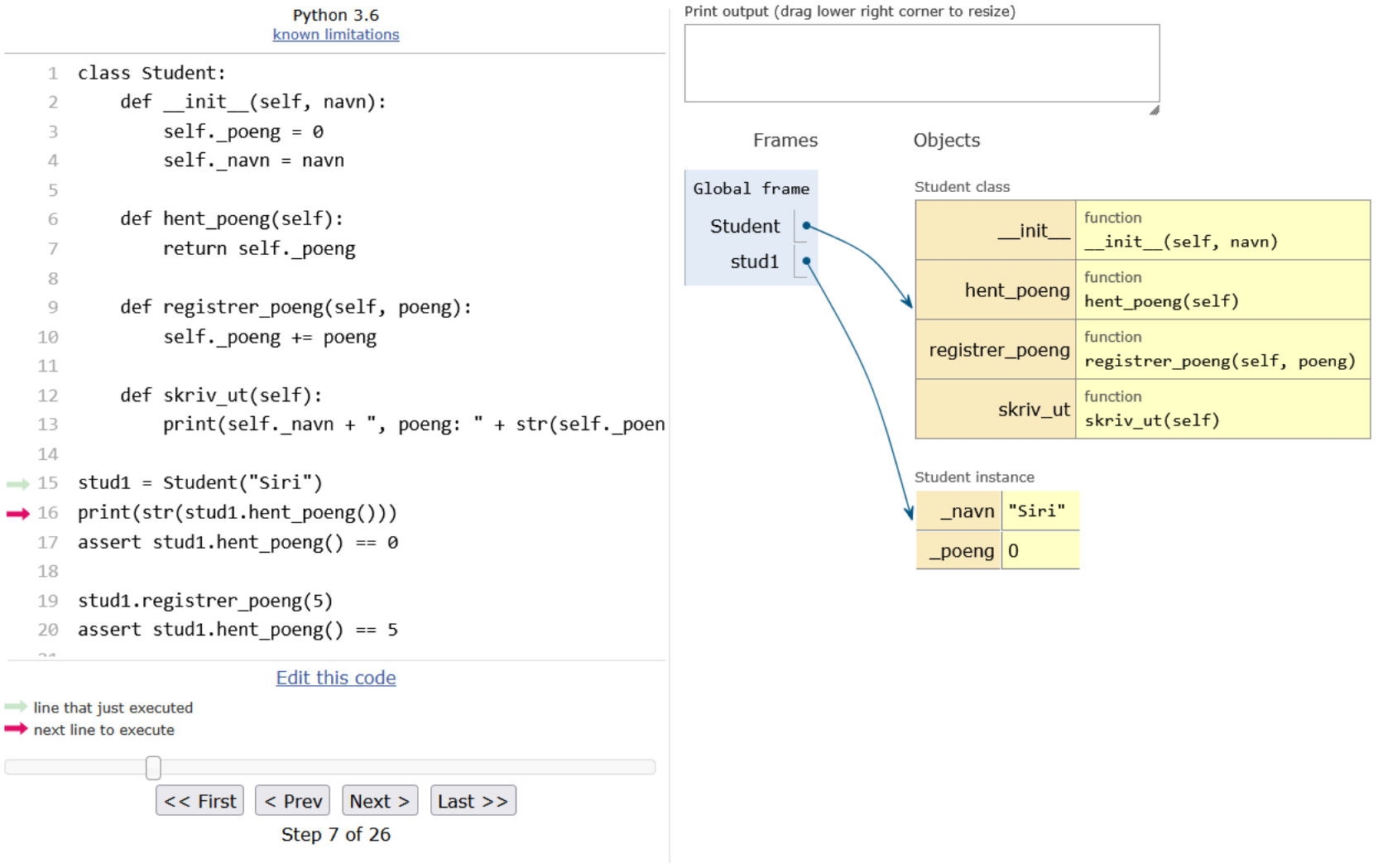
- Programmere må vite hvordan programmet *utføres*
  - Kodeflyt
  - Datastruktur og -representasjon

  (på et passende detaljnivå)

- Pythontutor og VSC debugger viser oss dette
  Nyttig for å forstå – og finne feil (debugging)

- Men kan bare vise abstraksjoner
  Det konkrete foregår inne i maskinen, som overganger mellom 0 og 1
  Innebærer noen valg av hva man viser, og hvordan!

Programtekst og utføring i Python Tutor



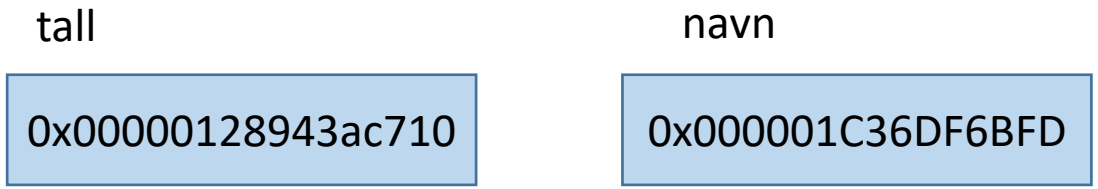
Kode: Student-program

Variabler i Python
Hva skjer når programmet utføres?

- En variabel kan tegnes som en "boks" som inneholder en verdi – et heltall, True eller False, eller et flyttall



- Ofte er det tilstrekkelig å tenke på det slik – Python skjuler detaljene for oss.
- MEN vi trenger et mer detaljert bilde når vi jobber med mer sammensatte objekter
- Variabelen inneholder ikke objektet selv – men adressen der objektet ligger i minnet



Variabler i Python

- Antakelig har du sett slike adresser allerede – om du har prøvd å printe ut et objekt av en klasse du har skrevet selv?

```
student = Student()
print(student)
print(id(student))
```

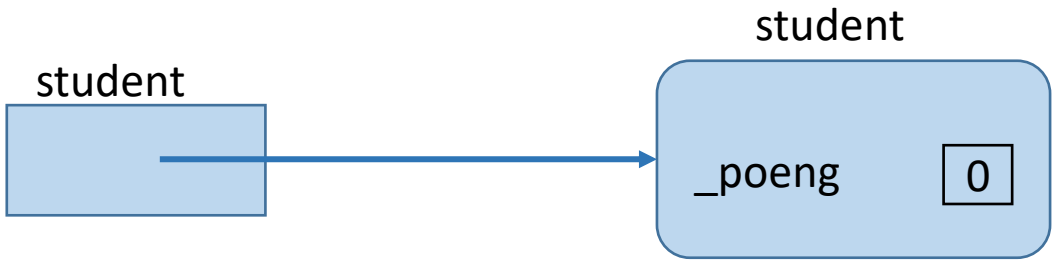
student

0x000002076E39A9E0

```
<__main__.Student object at 0x000002076E39A9E0>
2230937299424
```

Referanser til objekter

- Adressen til et objekt kaller vi en *objektreferanse* (*referanse*)

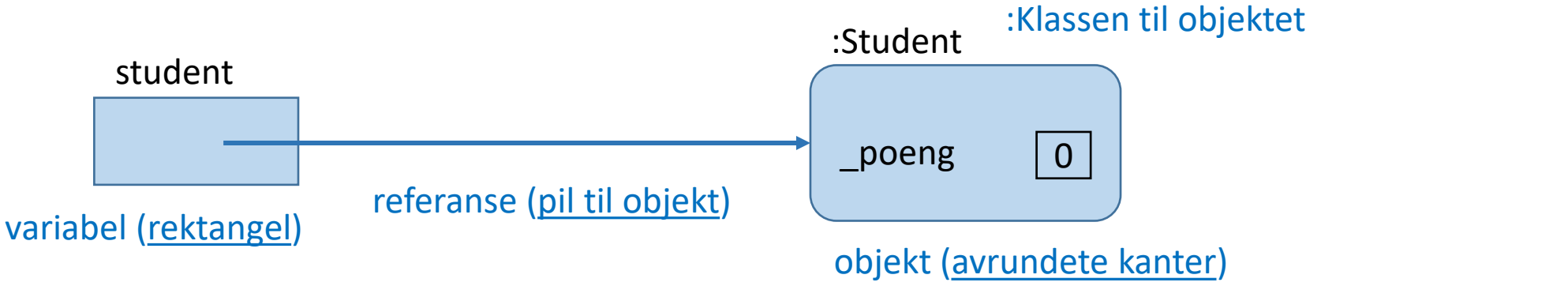


- Tegnes typisk som en pil som starter i en variabel og peker til objektet.

- Tegningen er nyttig for å minne oss på at *referansen og objektet er to ulike ting*

Datastrukturen under kjøring

Dette er **øyeblikksbilder**, fra **kjøring** av programmet



- Vi vil fortsatt angi verdien direkte inne i variabelen for “enkle” typer som int, boolean etc.



Referanser til objekter



- Kan lagres i en variabel, eller returneres av en funksjon/ metode
- Gjør det mulig å ta vare på, finne og bruke objekter når vi trenger dem
- Selve objektet kan lagres "hvorsomhelst" i minnet, og være stort eller lite – variabelen trenger bare plass til en adresse
- Referansen brukes for å kalle på metoder i objektet:

  ref.metode()

Forskjellen på referansen og objektet

- Når endrer vi referansen i en variabel, og når endrer vi (innholdet i) objektet?

```
student = Student()          # Her endres innholdet i variabelen
                             # (den refererer nå til det nye objektet)


student.registrer_poeng(5)   # Her endres innholdet i objektet
                             # variabelen refererer til ("peker på")
```

- Når vi snakker om *innholdet* i et objekt skjer det alltid med "dot-notasjon"
  - Enten utenfor klassen: **`student.registrer()`**
  - Eller i metodene inne i klassen: **`self._poeng += antall`**

Har du opplevd dette?

```python
liste1 = [1, 2, 3]
liste2 = liste1

liste1.append(4)
print(liste1)

print(liste2)
```

- Vi kopierer en liste-variabel til en annen.
- Endrer den opprinnelige listen
- Hva skjer med kopien?

```
PS M:\Ifi\Undervisning\IN1000 H25\kode> python .\mutable.py
[1, 2, 3, 4]
[1, 2, 3, 4]
PS M:\Ifi\Undervisning\IN1000 H25\kode>
```

- Kopien er også endret

- Merk at en referanse sendt inn som argument kopieres til parameteren
- Om objektet endres gjennom referansen i parameteren
- ..er dette altså samme objekt som argumentet refererer til på kallstedet

Metoder for strenger

Hva gjør metode-kallene **strip()** og **lower()** på en streng?

Hvordan pleier vi å kalle på streng-metoder?

```
min_streng = "Hei!  "

min_streng.strip()
min_streng.lower()
min_streng = min_streng.lower()
```

Strenger endres aldri – de er *immutable*

- Metoder for å endre en streng endrer aldri streng-objektet
- I stedet er de laget som funksjonsmetoder som **returnerer en ny streng**
- Denne returverdien må vi ta vare på for å bruke det nye objektet

```
navn = "Erna Solberg"

# Strenger er immutable, returnerer en endret kopi
navn_med_små_bokstaver = navn.lower()

print(navn)
print(navn_med_små_bokstaver)
```

```
PS M:\Ifi\Undervisning\IN1000 H25\kode> python .\immutable.py
Erna Solberg
erna solberg
PS M:\Ifi\Undervisning\IN1000 H25\kode>
```

Oppgave: Klassen Rektangel

Rektangler er nyttige elementer i mange sammenhenger

=> Mulighet for gjenbruk

Skriv en klasse Rektangel som

- Har en konstruktør som tar to parametere: **lengde** og **bredde**
- En metode **areal** som returnerer rektangelets areal
- En metode **endre** som endrer størrelsen på rektangelet og har to parametere - lengde og bredde som angir hvor mye sidene i rektangelet skal endres med

Klassen Rektangel

```
class Rektangel:
  def __init__(self, lengde, bredde):
    self._lengde = lengde
    self._bredde = bredde

  def areal(self):
    return self._lengde*self._bredde

  def endre (self, lengde, bredde):
    self._lengde += lengde
    self._bredde += bredde
```

Hva er self?

- Vi kan lage mange objekter av en klasse – alle har hver sin kopi av instansvariablene

- self-parameteren til konstruktøren og andre metoder holder rede på **hvilket objekt sine instansvariabler vi jobber med**

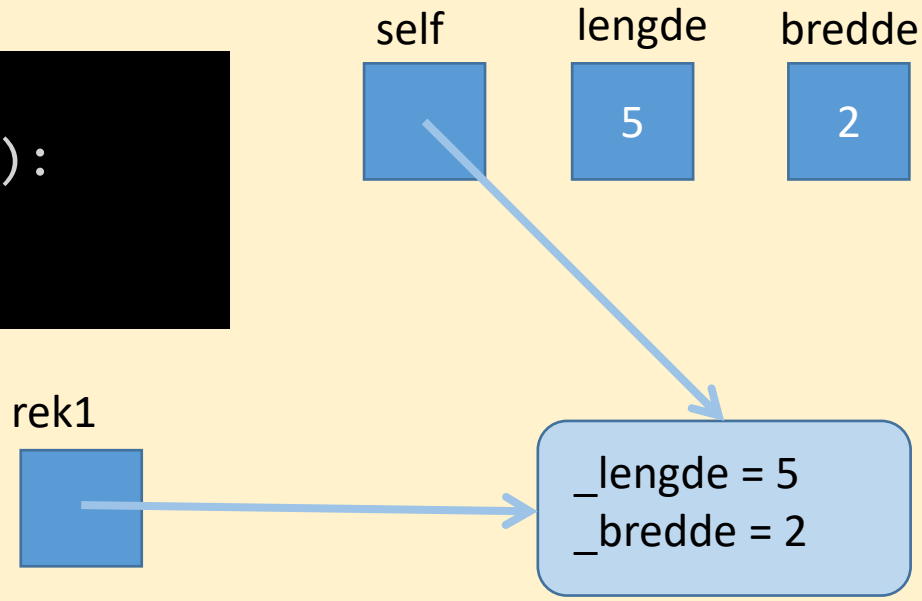
➢ I konstruktøren får self referansen til det nye objektet som opprettes

➢ Inne i en metode får self samme verdi som objektreferansen metoden er kallt på (variabelen før ‘.’)

Hva skjer når vi oppretter objekter?

```
class Rektangel:
  def __init__(self, lengde, bredde):
    self._lengde = lengde
    self._bredde = bredde
```

```
rek1 = Rektangel(5, 2)
```

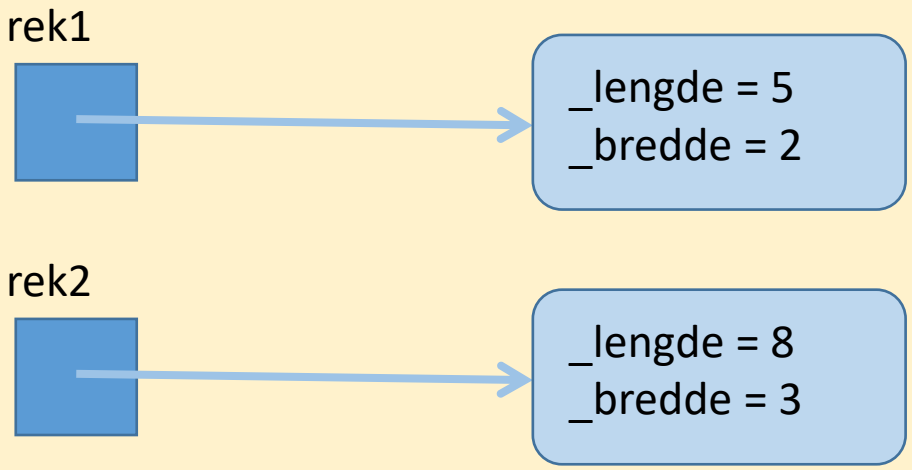


Hva skjer når vi oppretter objekter?

```
class Rektangel:
  def __init__(self, lengde, bredde):
    self._lengde = lengde
    self._bredde = bredde
```

```
rek1 = Rektangel(5,2)

rek2 = Rektangel (8,3)
```



Hva skjer når vi kaller metoder?

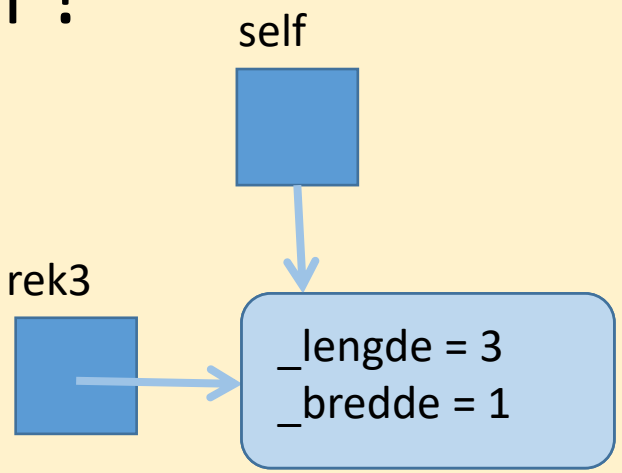
```
class Rektangel:
  def __init__(self, lengde, bredde):
    self._lengde = lengde
    self._bredde = bredde

  def areal(self):
    return self._lengde*self._bredde

  def endre(self,lengde,bredde):
     self._lengde +=lengde
     self._bredde += bredde
```

```
rek3 = Rektangel(5,2)
print (rek3.areal())

rek3.endre(-2,-1)
print (rek3.areal())
```



Kjøring

```
M:> python testrektangel.py
10
3
M:>
```

Metoder kan endre, lese og bruke egenskapene til objektet de kalles på

```
rek3 = Rektangel(5,2)
print (rek3.areal())

rek3.endre(-2,-1)
print (rek3.areal())
```

I metoden **areal** gjør objektet en beregning for oss som vi bruker på kallstedet

Metoden **endre** endrer egenskapene i objektet, som forblir endret etter metoden avsluttes

Nytt kall på **areal** gir en annen verdi fordi objektets egenskaper er endret

Lokale variabler i metoder

- Kan ikke aksesseres utenfor egen metode (i motsetning til instansvariablene)
- Oppstår og dør hver gang metoden kalles (som i funksjoner)

```
class Rektangel:
  def __init__(self, lengde, bredde):
    self._lengde = lengde
    self._bredde = bredde

  def areal(self):
    beregnet_areal = self._lengde*self._bredde
    return beregnet_areal
```

- Parametere oppfører seg som lokale variabler – MEN initialiseres med argumentene i kallet (samme som i funksjoner)

Lite sideblikk til skop

(bare for de som har stilt seg spørsmålet!)

Fra uke 6 om skop:

1. Funksjoner deler data kun gjennom parameteroverføring og returverdier – veldefinerte grensesnitt gir oversikt og reduserer feil
2. For å unngå feil med navnekollisjoner/ forvekslinger bruker vi ikke globale variabler

Hva med metoder i en klasse - som deler instansvariabler??

1. OK å dele data på tvers av metoder *innenfor* en klasse, her har vi oversikt og kontroll
2. Alle instansvariabler må aksesseres med prefix `self`.
   - Vi angir alltid (med .-notasjon) argumentet til self i metodekallet

   => En instansvariabel kan ikke forveksles med en lokal variabel (eller parameter)

   $\Rightarrow$ En instansvariabel kan ikke forveksles med en instansvariabel i et annet objekt

Verdien `None`

- Verdien `None` signaliserer at en referansevariabel ikke holder på noe objekt i øyeblikket.
- `None` kan *i noen tilfeller* være en nyttig initialverdi
- Kan også være nødvendig å sjekke mot `None` for å unngå å kalle på en metode i et objekt som ikke finnes (gir ellers feil under kjøring):

-
```
if student is not None:
    student.registrer_poeng(5)
```

- Mer om `None`, `is`, `is not` og *sammenligning* av referanser og objekter neste uke!

Eksempel: Navn

- Vi håndterer veldig ofte navn i programmene våre – kan vi lage noe som hjelper oss med å sammenligne og kjenne igjen navn i ulike formater?
- Skriv en klasse Navn som skal huske og presentere på flere formater navn bestående av fornavn, mellomnavn og etternavn

- Uformelt grensesnitt
  - Konstruktør m/ parametere for for-, mellom- og etternavn
  - Metoder for å
    - hente på form egnet for sortering (Støre, Jonas Gahr)
    - hente naturlig (Jonas Gahr Støre)

Klassen Navn - grensesnitt

```
class Navn:
  # Konstruktør, tar argumenter til 3 navnedeler
  # instansvariabler for for-, mellom- og etternavn
  def __init__(self,fornavn,mellom,etter):

  # returnerer en string med navnet på sorterbar form
  def sorterbar(self):

  # returnerer en string med navnet på naturlig form
  def naturlig(self):
```

Kode

Klassen Navn

Filen navn.py

```
class Navn:
  def __init__(self,fornavn,mellom,etter):
    self._fornavn = fornavn
    self._mellom = mellom
    self._etter = etter

  def sorterbar(self):
    alfabetisk_navn = self._etter + ", " + self._fornavn + " " + self._mellom
    return alfabetisk_navn

  def naturlig(self):
    naturlig_navn = self._fornavn + " " + self._mellom + " " + self._etter
    return naturlig_navn
```

Klassen Navn: Testprogram

```python
from navn import Navn

def hovedprogram():
    navn1 = Navn("Siri", "Moe", "Jensen")
    navn2 = Navn("Geir", "Kjetil", "Sandve")

    print (navn1.sorterbar())
    print (navn2.sorterbar())
    print (navn2.naturlig())

hovedprogram()
```

```
Jensen, Siri Moe
Sandve, Geir Kjetil
Geir Kjetil Sandve
```

Fil: test_navn.py

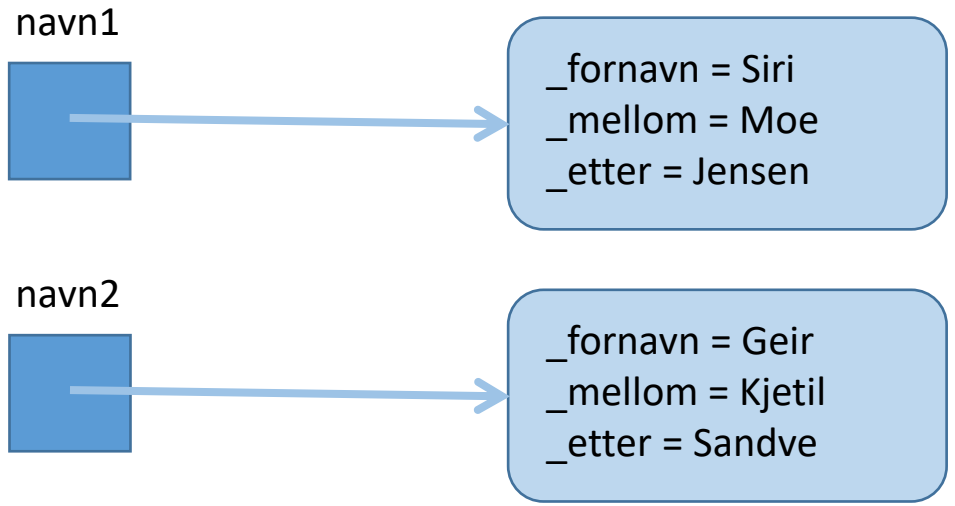
Klassen Navn: Testprogram

```
from navn import Navn

def hovedprogram():
    navn1 = Navn("Siri", "Moe", "Jensen")
    navn2 = Navn("Geir", "Kjetil", "Sandve")

    print (navn1.sorterbar())
    print (navn2.sorterbar())
    print (navn2.naturlig())

hovedprogram()
```




```
Jensen, Siri Moe
Sandve, Geir Kjetil
Geir Kjetil Sandve
```

Hva slags klasse er Navn?

Kan du gjøre endringer i et Navn-objekt?

- Klassen Navn er *immutable* – "uforanderlig", akkurat som strenger
- Unngår feil som følge av at flere variabler refererer til samme objekt

```
class Navn:
  def __init__(self,fornavn,mellom,etter):
    self._fornavn = fornavn
    self._mellom = mellom
    self._etter = etter

  def sorterbar(self):
    alfabetisk_navn = self._etter + ", " + self._fornavn + " " + self._mellom
    return alfabetisk_navn

  def naturlig(self):
    naturlig_navn = self._fornavn + " " + self._mellom + " " + self._etter
    return naturlig_navn
```

Flere objekter av samme klasse

- En klasse er et mønster å lage objekter av
- Nyttig for å representere mange "ting" som følger samme mønster, f. eks. navn
- Skal lese inn et ukjent antall navn, opprette navneobjekter, legge dem i en liste og deretter skrive dem ut på *sorterbar* form (men ikke sortert)

Fil: test_flere_navn.py

Flere objekter av samme klasse

- En klasse er et mønster å lage objekter av
- Nyttig for å representere mange "ting" som følger samme mønster, f. eks. navn
- Skal lese inn et ukjent antall navn, opprette navneobjekter, legge dem i en liste og deretter skrive dem ut på *sorterbar* form (men ikke sortert)

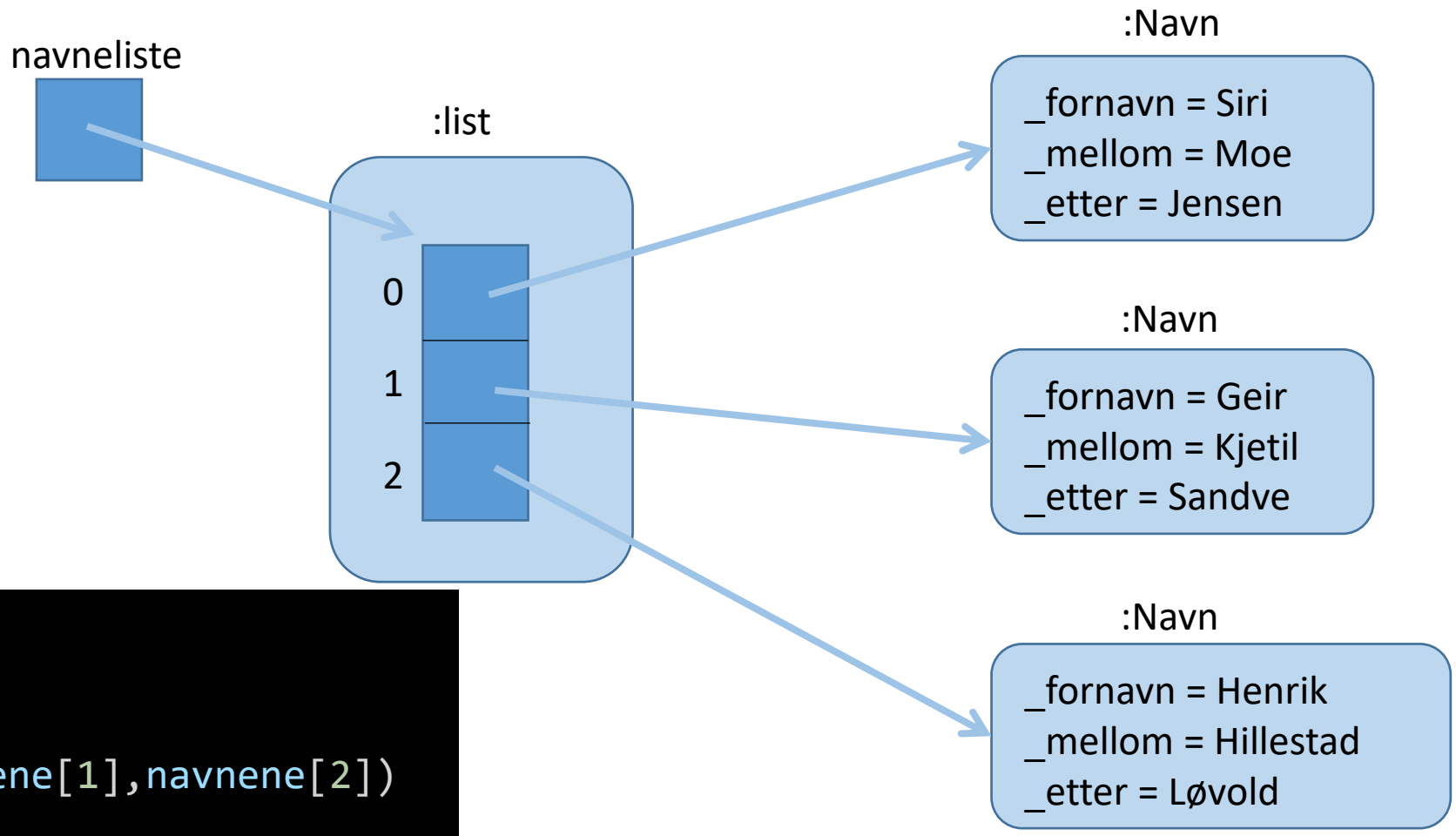
```
from navn import Navn

navneliste = []
les = input("Oppgi navn på naturlig form: ")
while les != "" :
    navnene = les.split()
    nytt = Navn(navnene[0],navnene[1],navnene[2])
    navneliste.append(nytt)
    les = input("Oppgi navn på naturlig form: ")

for et_navn in navneliste:
    print(et_navn.sorterbar())
```

Fil: test_flere_navn.py

Datastrukturen etter innlesing av tre navn




```
..
while les != "" :
    navnene = les.split()
    nytt = Navn(navnene[0],navnene[1],navnene[2])
    navneliste.append(nytt)
    les = input("Oppgi navn på naturlig form: ")
..
```

Eksempel: Klasse Person

- Skal lagre og skrive ut data om en person
  - Navn
  - Alder
  - (og mye mer etter hvert)

Implementasjon av Person

- Hvordan representere navnet til personen i klassen?
- Kan ha en instansvariabel som inneholder en tekst (string)

  ... men da vet vi ingenting om hvordan navnet er bygd opp

$\Rightarrow$ velger å bruke vår klasse Navn som en mer intelligent og fleksibel representasjon av personnavn

- Hvert Person-objekt får en instansvariabel som refererer et Navn-objekt
- Da kan et Person-objekt oppgi navnet sitt på flere former!

Implementasjon av Person

Oppgave: Fyll ut konstruktøren og skriv_ut-metoden

Filen person.py

```
class Person:
  def __init__(self,fullt_navn,alder):
    self._navn = fullt_navn
    self._alder = alder

  def skriv_ut(self):
    print ("Navn: " + self._navn.naturlig() )
    print ("Alder: " + str(self._alder))
```

- Objekter kan ha instansvariabler som refererer til andre objekter
  - f eks en liste, eller
  - et objekt av en annen klasse, eller
  - et objekt av samme klasse som objektet selv (=> uke 10)

Klassen Person: Testprogram

Fil: test_person.py

```
from navn import Navn
from person import Person

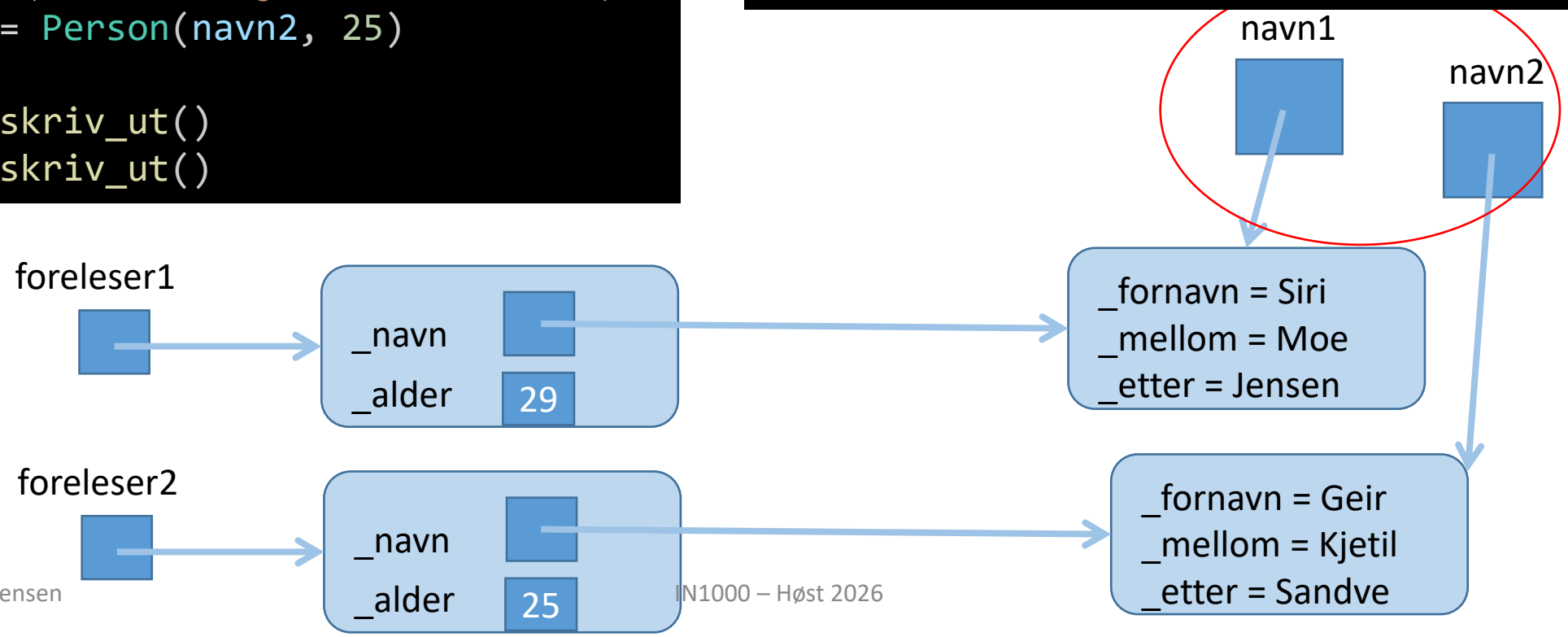
navn1 = Navn("Siri", "Moe", "Jensen")
foreleser1 = Person(navn1, 29)

navn2 = Navn("Geir", "Kjetil", "Sandve")
foreleser2 = Person(navn2, 25)

foreleser1.skriv_ut()
foreleser2.skriv_ut()
```

```
from navn import Navn
class Person:
  def __init__(self,fullt_navn,alder):
    self._navn = fullt_navn
    self._alder = alder

  def skriv_ut(self):
    print ("Navn: " + self._navn.naturlig() )
    print ("Alder: " + str(self._alder))
```



Testprogram: Flere personer

fil: test_flere_personer.py

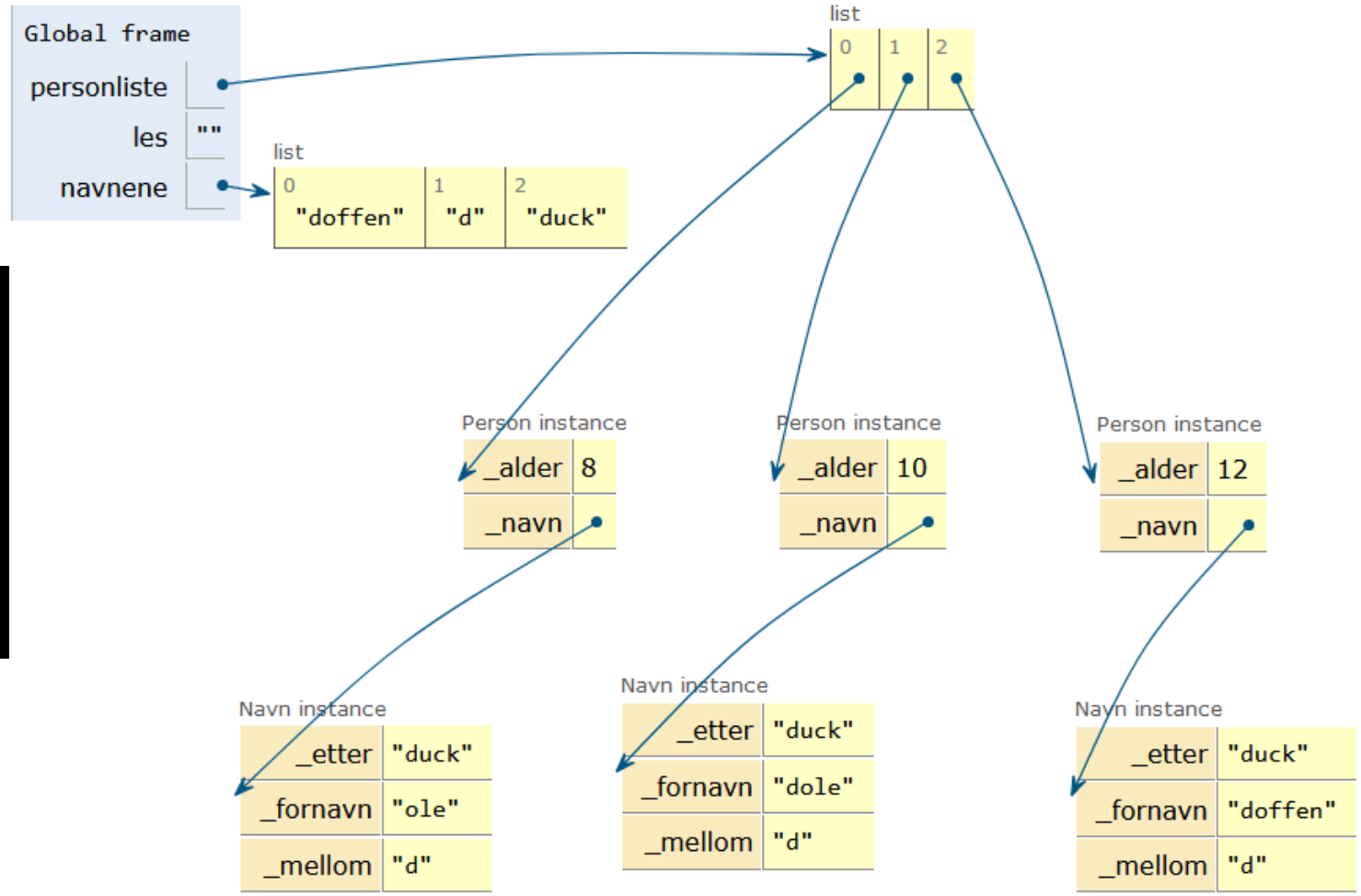
```
from navn import Navn
from person import Person

personliste = []
les = input("Oppgi navn på naturlig form: ")
while les != "" :
    navnene = les.split()
    nytt = Navn(navnene[0],navnene[1],navnene[2])
    alder = int(input("Oppgi alder: "))
    ny_person = Person(nytt, alder)
    personliste.append(ny_person)
    les = input("Oppgi navn på naturlig form: ")

for person in personliste :
    person.skriv_ut()
    print("\n")
```

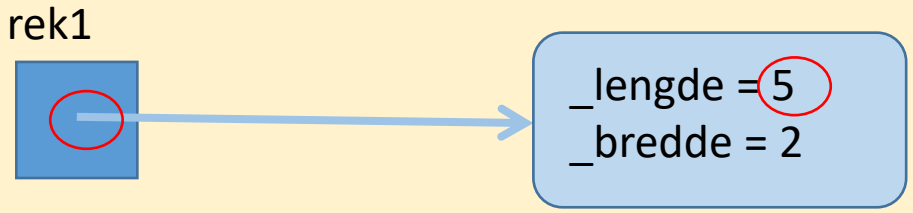
Visualisering i Pythontutor (forenklet)

```
Oppgi navn på naturlig form: ole d duck
Oppgi alder: 8
Oppgi navn på naturlig form: dole d duck
Oppgi alder: 10
Oppgi navn på naturlig form: doffen d duck
Oppgi alder: 12
Oppgi navn på naturlig form:
```



Viktigste i dag

- En referanse er en “adresse” til et objekt – den er ikke objektet
- Å endre innholdet i en referansevariabel (til en ny referanse) er noe annet enn å endre objektet variabelen refererer til



- Hjelpemidler:
  - Visualiser! Tegn selv, bruk Pythontutor
  - Tenk uttrykk! Hvilken rekkefølge evalueres deluttrykk i, hvilken verdi evaluerer hvert deluttrykk til, og hvordan brukes denne verdien videre?
  - Mens du løser mange oppgaver ☺

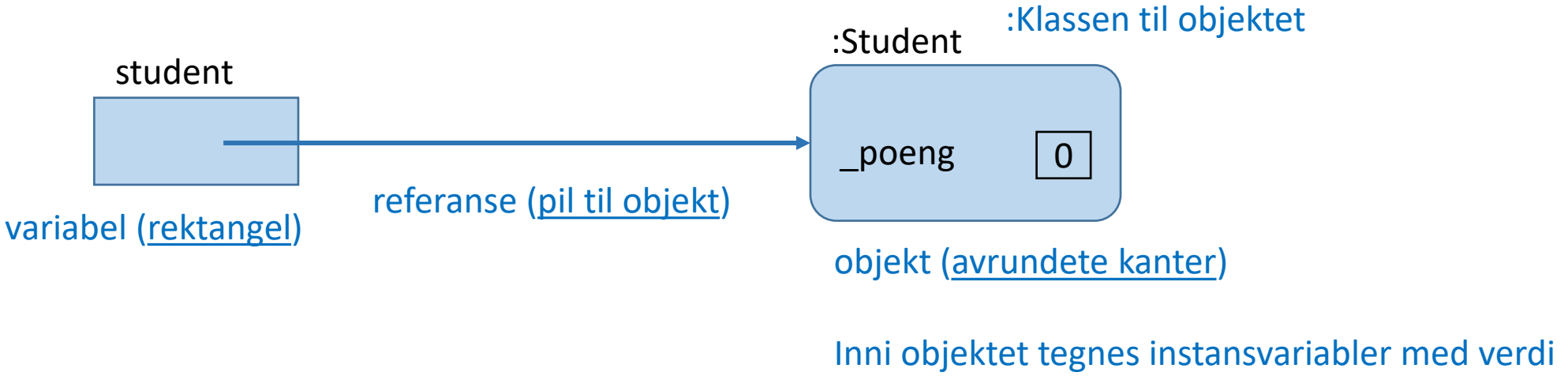
Tegningene er *modeller*

- Inneholder ikke (alltid) all informasjon
- Vi tar med det vi ønsker å fokusere på (forstå, få oversikt over)
- Handler om å sette sammen enklere elementer i større sammenhenger

- Trenger en notasjon som formidler informasjon (skille på variabel, objekt, referanse – hvilke klasse tilhører objektet)
- Notasjonen må være kjent for de som skal bruke tegningen – du selv, eller andre
- Skal brukes i oblig 7

Dette er **øyeblikksbilder**, fra **kjøring** av programmet

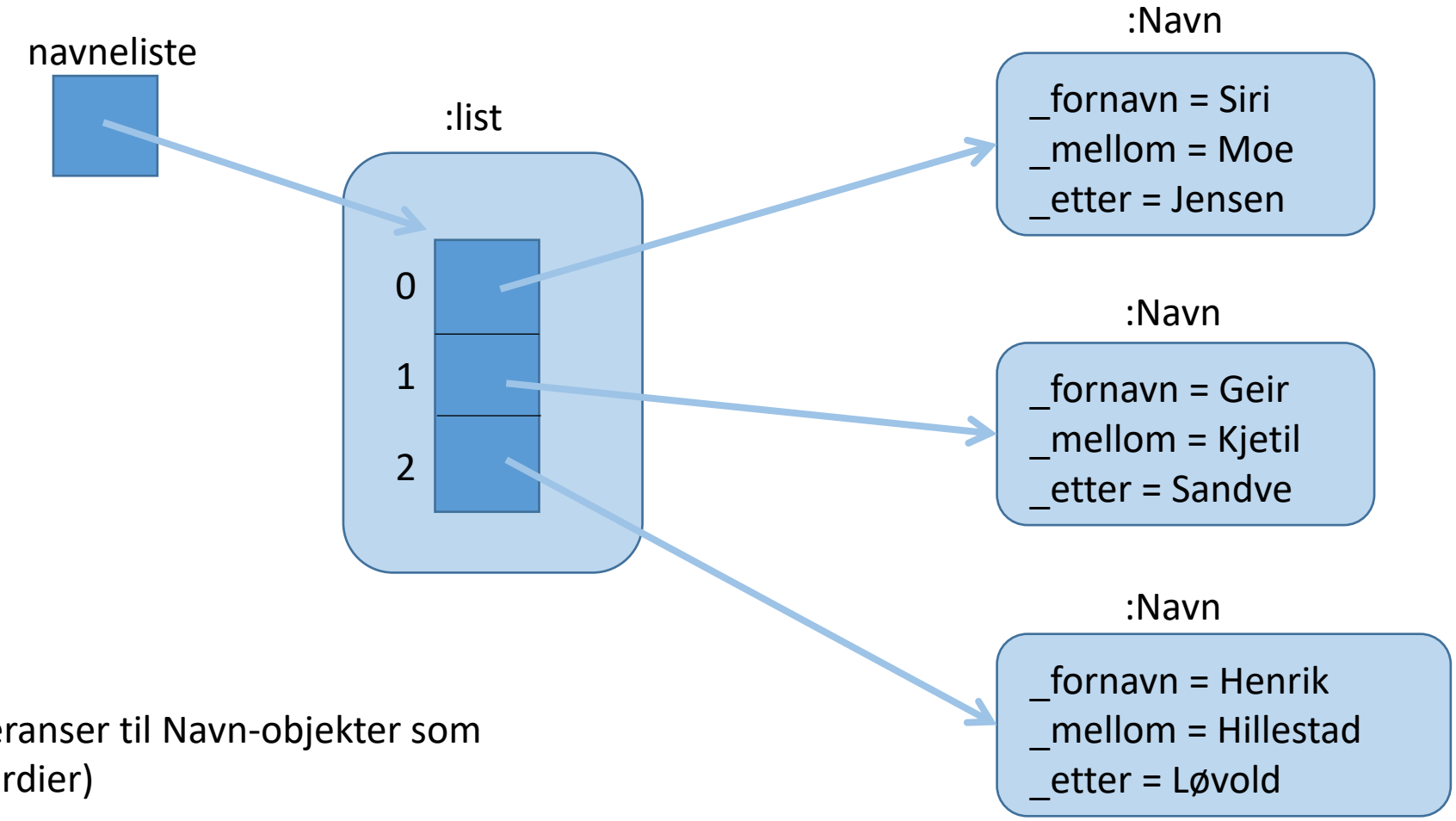
Datastrukturen under kjøring



- Vi vil fortsatt angi verdien direkte inne i variabelen for "immutable" typer som int, boolean, string

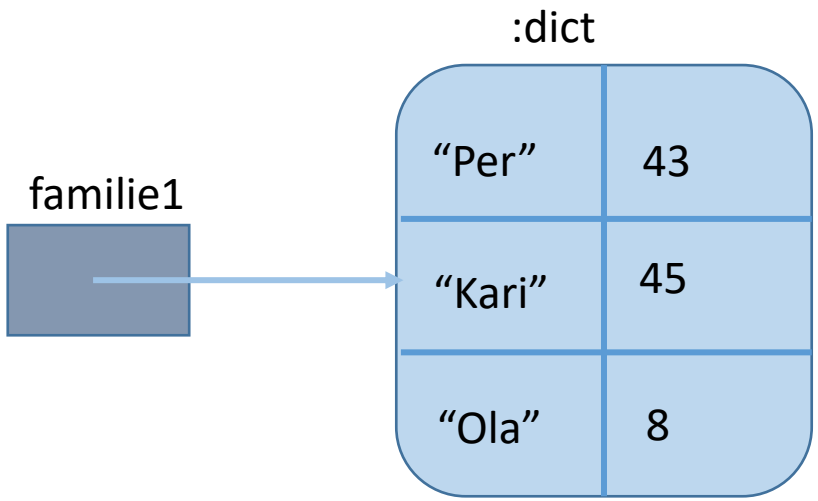


Visualisering - liste



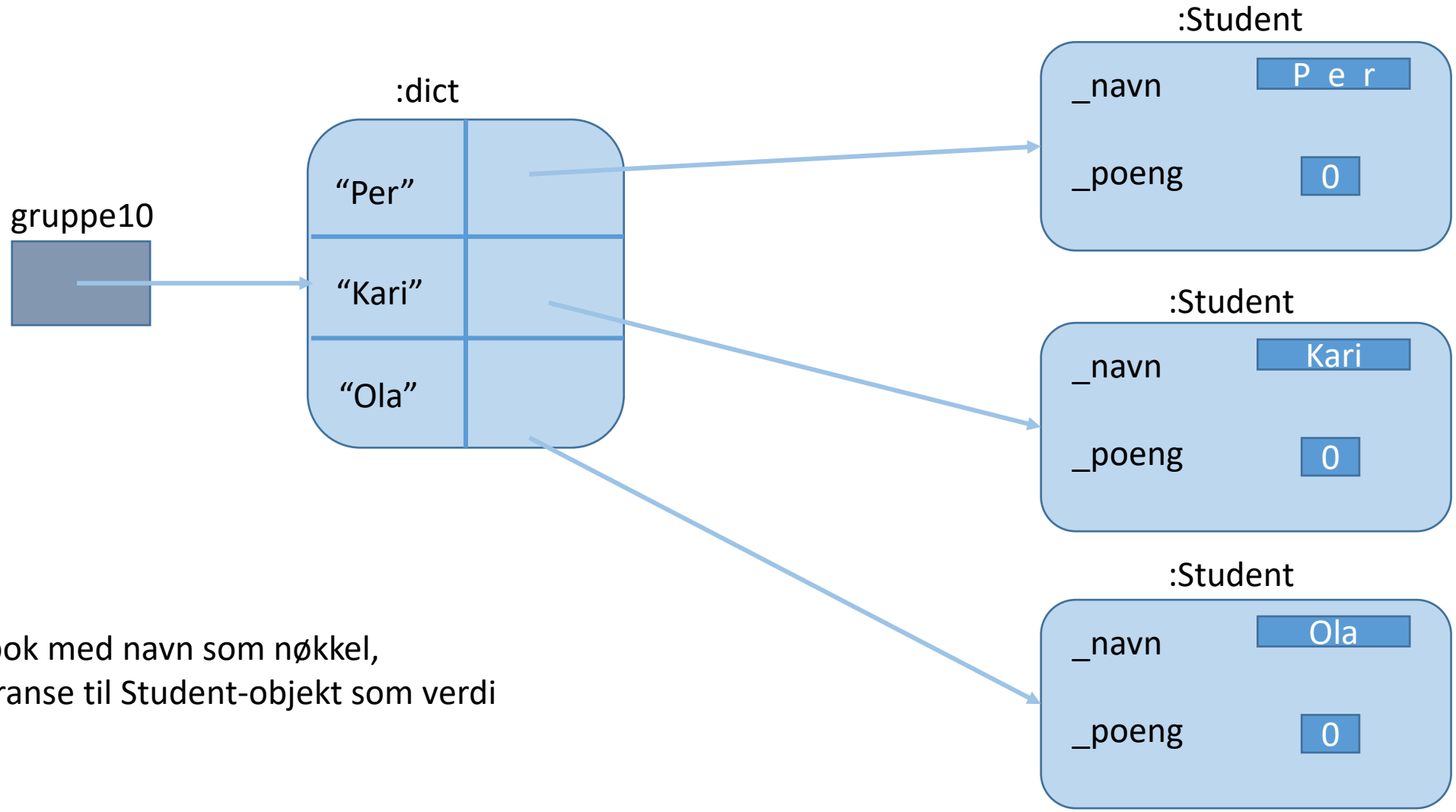
Liste med referanser til Navn-objekter som elementer (verdier)

Visualisering - ordbok



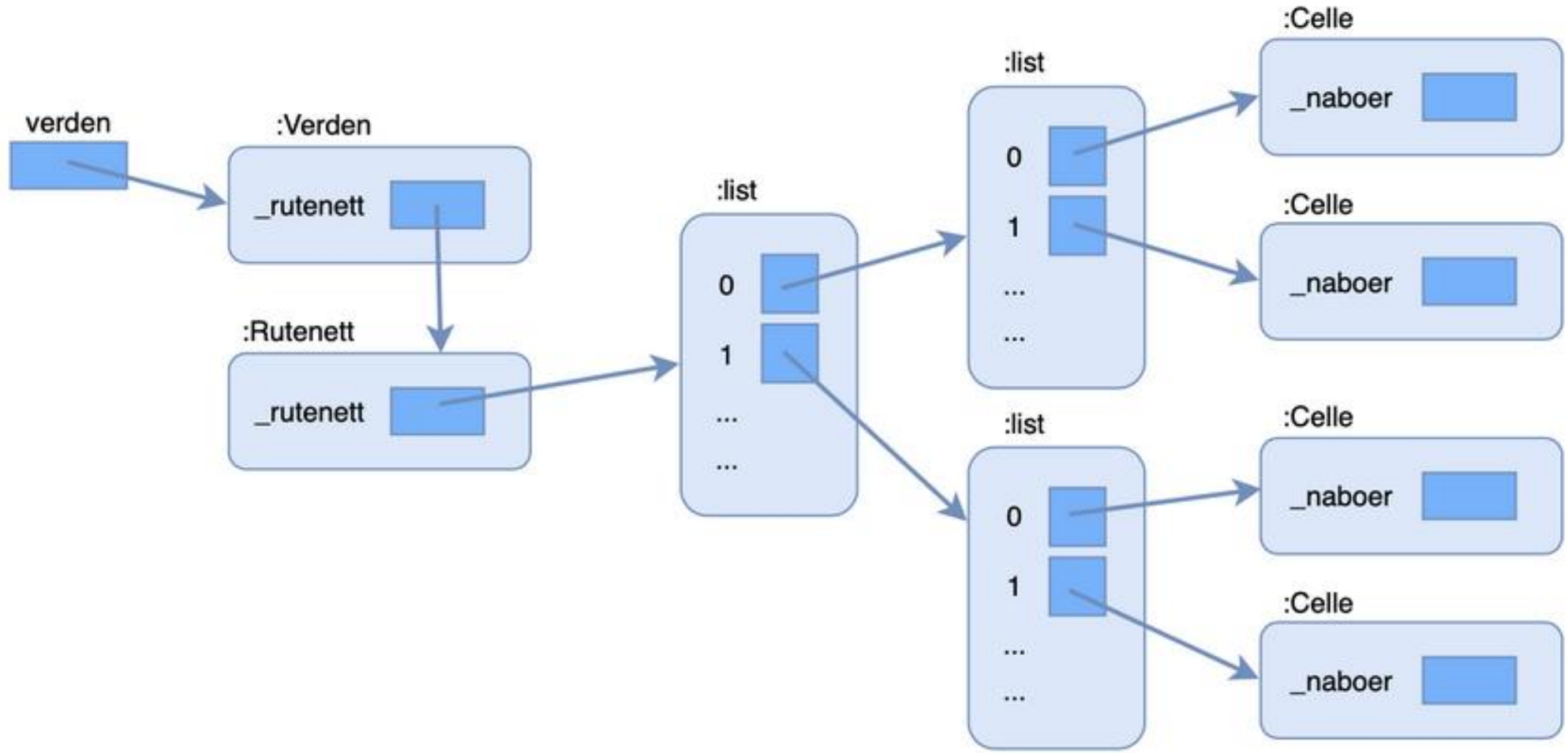
Ordbok med navn som nøkkel, alder som verdi

Visualisering - ordbok



Ordbok med navn som nøkkel,
Referanse til Student-objekt som verdi

Visualisering av komplekse datastrukturer



Innlevering 7: Alt obligatorisk, større program

- Frist 20.10.
  Kan få nytt forsøk om det ikke er altfor store mangler ("gjort et reelt forsøk")
- NB: Klassen Film bør skrives og testes ferdig denne uken!

- Program med to klasser og flere objekter, og fylle ut to test-programmer som sjekker at hver metode i klassene fungerer som de skal
- I innlevering 8 og på eksamen vil du skrive et (enda) større objektorientert program

- Obligatorisk oppgave 7 gir mulighet for å jobbe grundig med dette stoffet.

- Bruk gruppetimene og Trix-oppgaver, les tilbakemeldinger på oblig 6

Læringsmål uke 8

- Forstå (mer av) hva som skjer bak kulissene når vi oppretter og bruker objekter
- Kjenne til forskjellen på å endre en referansevariabel og å endre objektet den refererer til
- Kunne skrive programmer med samlinger av (referanser til) objekter
- Kunne forstå og tegne datastruktur-tegninger
- Kunne sette seg inn i og skrive mindre programmer med flere klasser, og med objekter som refererer til andre objekter

Neste uke

- "Magiske (spesielle) metoder" for egendefinerte klasser
  - Hvordan representere objekter med innhold som en streng?
  - Hvordan sjekke om to objekter er "like"?

- Objekter i ulike strukturer
  - Samlinger av objekter
  - Referanser mellom objekter av ulike klasser