Objektorientert programmering i Python

IN1000

Høst 2021 – uke 8

Siri Moe Jensen

Å lære objektorientert programmering

- En ny måte å tenke på
- Alt foregår "inne i maskinen" og "inne i vårt hode"
- Kan kjennes veldig abstrakt og uhåndterlig
- Nye begreper, konsepter, og språk-elementer
- Og grunnelementene fra uke 1-6 sitter kanskje ikke 100% - ennå

⇒ Gi det tid!

- Se på eksempler, løs mange oppgaver selv
- Lær mønstrene (syntaks) og begrepene, aksepter at ikke alt kan "forstås" umiddelbart
- Bruk læreboken, gå på grupper (gjerne flere), se på livekoding og løs oppgaver
- Tegn selv – og bruk gjerne Pythontutor.org

⇒ De mentale modellene av hva som foregår kommer etter hvert som du jobber med stoffet

Dette skal vi bruke de neste 1-2 månedene på!

Plan for i dag

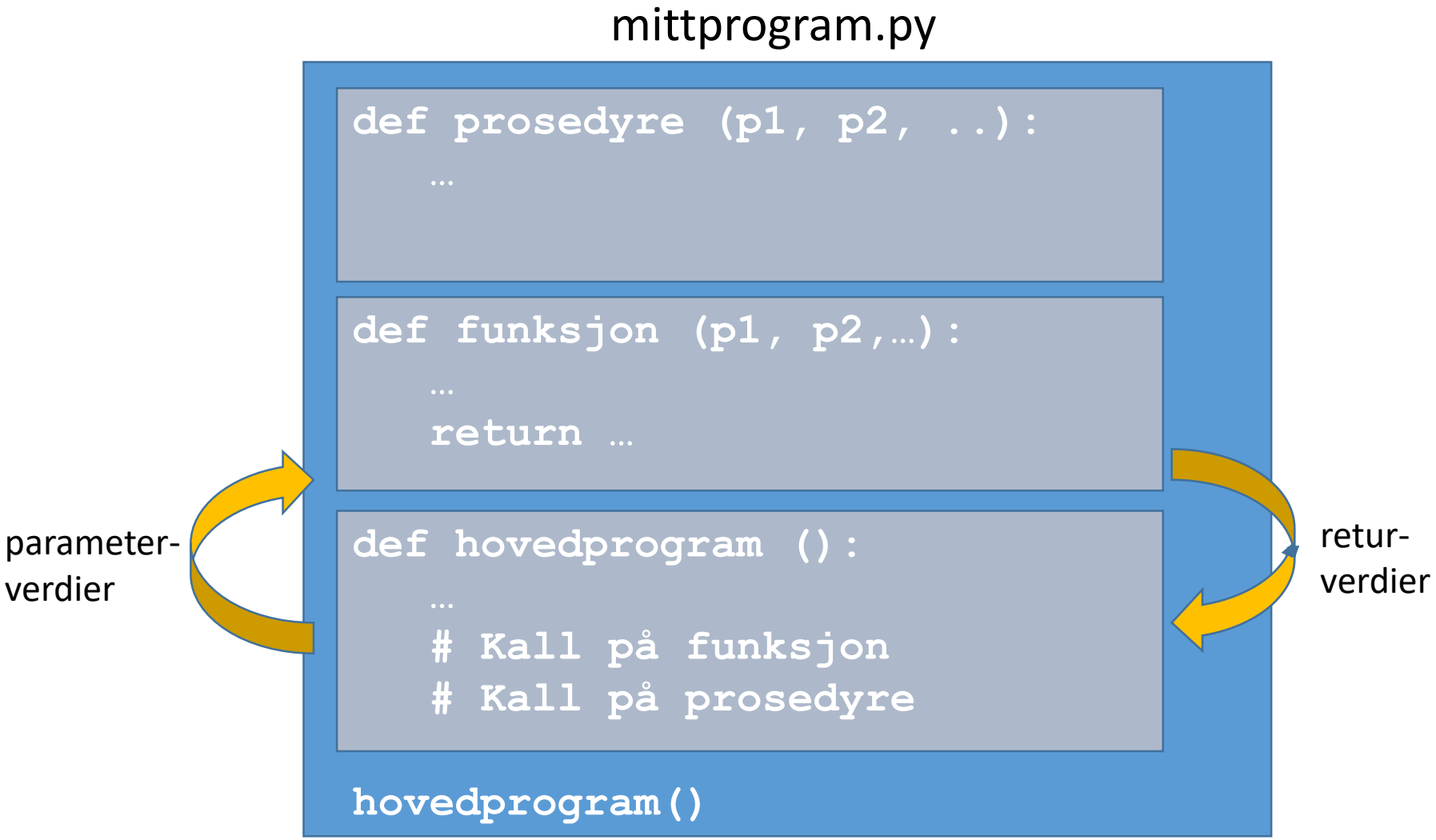
- Repetisjon og gjennomgang av forrige ukes Menti-oppgaver
- Referanser og objekter
- "Uforanderlige" (immutable) objekter

- Eksempel – hva skjer i detalj når vi oppretter objekter og kaller metoder?
- Organisering av mange objekter
- Forsmak på neste uke:
  Flere klasser, objekter som refererer til andre objekter

Obligatoriske krav for å få gå opp til eksamen

- Både oblig 7 og oblig 8 må godkjennes for å gå opp til eksamen
- Se tekst og lenker på semestersiden (under Obligatoriske innleveringer) om krav til eget arbeid:
  - Samarbeid om teori og andre oppgaver – skriv egen kode for innlevering!
- Lever i god tid i tilfelle tekniske problemer eller annet. Hvis du leverer flere ganger er det siste versjon som gjelder.
- Behov for utsettelse eller nytt forsøk – se under Obligatoriske innleveringer

Program med prosedyrer og funksjoner



Program med klasser

mittklasseprogram.py

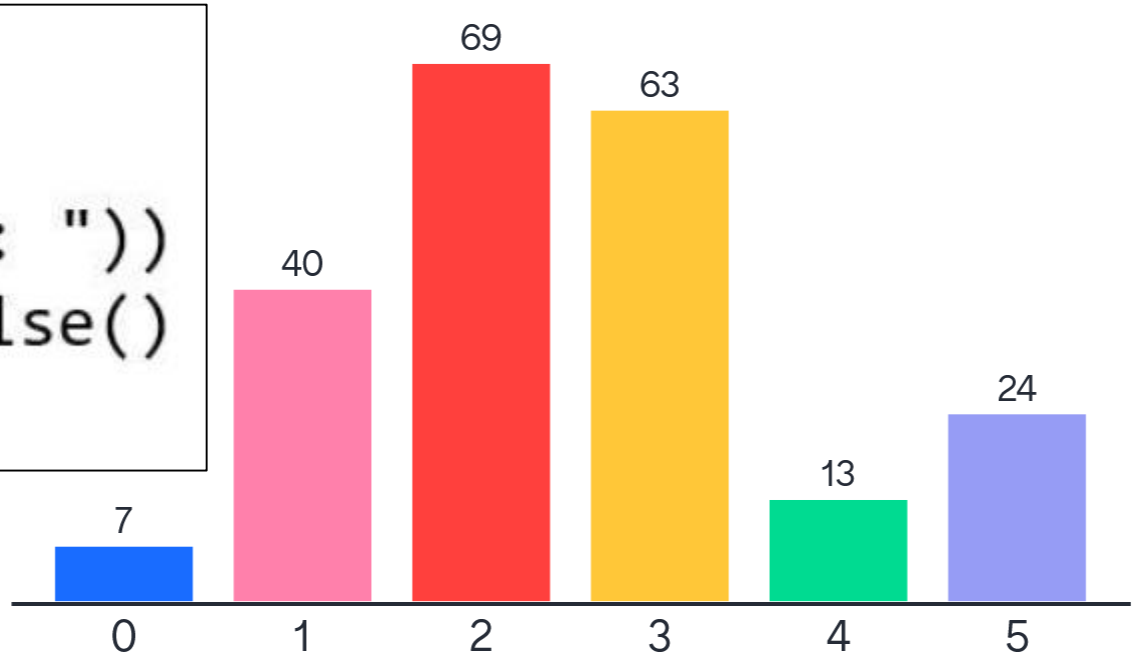
```
class Student:
    def __init__(self,p1):
        self._instvar1 = p1
        self._instVar2 = 3.14

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

```
def hovedprogram ():
    …
    # oppretter og bruker objekter
```

```
hovedprogram()
```

Hvor mange av disse linjene er et uttrykk (hele linjen)?

```
x * y + z
1
int(input("Alder: "))
stud.hentDeltakelse()
True
```



Voting is closed

199

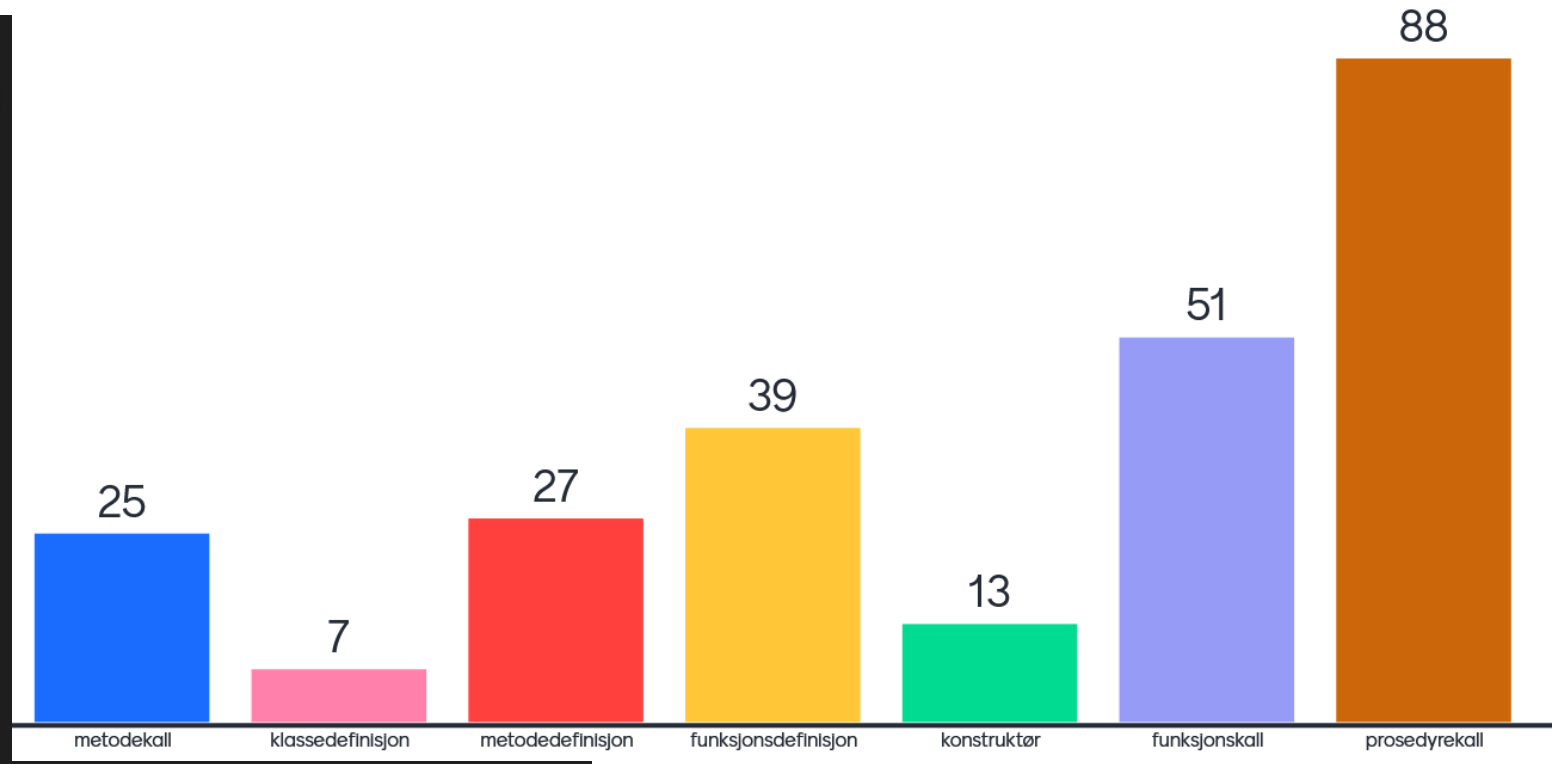
Alle disse linjene er uttrykk!

test f eks print(int(input("Alder: ")))

eller print(True)

Hva inneholder *ikke* dette programmet?

```python
class Rom:
  def __init__(self, nr, navn, opsys, ant):
    self._nr = nr
    self._navn = navn
    self._type = opsys
    self._ant = ant

  def hentAntall(self):
    return self._ant

  def hentNavn(self):
    return self._navn

  def endreOpsys(self, nyttOpsys):
    self._type = nyttOpsys

assembler = Rom(63417, "Assembler", "Linux", 40)
assembler.endreOpsys("Windows")
print(assembler.hentNavn(), "har", assembler.hentAntall(), "plasser")
```



- Har ingen funksjonsdefinisjon
- Har ikke funksjonskall om man ikke regner prosedyre som en type funksjon

Skriv linjen som mangler for at dette programmet virker

46 stk:
self._antSkiver = antall

```
class Matpakke:
    def __init__(self, antall):
        self._paalegg = ""

    def velgPaalegg(self):
        self._paalegg = input("Velg pålegg: ")

    def skrivMatpakke(self):
        print(self._antSkiver, "skiver med ", self._paalegg)

lunsj = Matpakke(3)
lunsj.velgPaalegg()
lunsj.skrivMatpakke()
```

Andre forslag (virker ikke):

```
self._antall = antall
self._antSkiver = ""
self._antSkiver = \
    input("velg antall skiver")
return self._paalegg
```

Enkle og mer sammensatte typer

Har brukt verdier av ulike *innebygde typer*. Enkle typer som

- Heltall (1, 45, -1)
- Sannhetsverdier (True, False)

og mer *sammensatte typer* som kan inneholde mer enn en verdi og tilbyr tjenester for å manipulere disse: Lister, ordbøker og strenger.

Alt dette er generiske og anvendelige typer helt uavhengig av hva slags programmer vi skriver og hva de skal brukes til, og som følger med alle Python tolker.

Eksempler på sammensatte typer

Lister som kan inneholde mange enkeltverdier og tilbyr tjenester som

- **append(element)**
- **pop()**

Strenger som består av 0 til mange tegn og kan manipuleres på ulike måter

- fjerne blanke med **strip()**
- gjøre om til små bokstaver med **lower()**

NB: Endrer vi strengen – eller lager vi en ny?
Mer om dette senere.

Filobjekter som lar oss åpne og lukke, lese og skrive på en fil

Vi kan lage våre egne typer!

- Ved å definere *våre egne klasser* kan vi selv «konstruere» slike sammensatte – men mer skreddersydde – typer.
- Vi kan deretter opprette og bruke ett eller flere objekter av hver type.
- Dette er nyttig i veldig mange sammenhenger!
- Forrige forelesning laget vi klassene Student og Navn

Eksempler fra uke 7

```
class Student:
    def __init__(self, navn):
        self._antDeltatt = 0
        self._navn = navn

    def registrer(self):
        self._antDeltatt += 1

    def hentAntDeltakelse(self):
        return self._antDeltatt

    def hentNavn(self):
        return self._navn
```

Spørsmål: Kan vi endre på et Navn-objekt etter at det er opprettet?

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

  def sortert(self):
    alfNavn = self._etter + ", " + self._fornavn +
              " " + self._mellom
    return alfNavn

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

Så hva er en klasse?

- Et mønster
- Kan sammenlignes med en pepperkakeform
- Selv om vi har en klassedefinisjon i programmet vårt, har vi ingen data – og kan heller ikke gjøre noe med dem

=> Vi har kakeformen, men ingen pepperkaker..

..før vi oppretter *objekter* av klassen

- Ulike klasser (kakeformer) lager ulike objekter (pepperkaker)
- Vi kan lage så mange objekter (pepperkaker) vi trenger av en klasse (form)

Å definere en klasse, opprette objekter og kalle på metoder

Repetisjon

```
class Student:
```
klassenavn

```
  def __init__(self):
    self._antDeltatt = 0
```
konstruktør

```
  def registrer(self):
    self._antDeltatt += 1
```
metode

```
  def hentDeltakelse(self):
    return self._antDeltatt
```
(funksjons-)metode

```
nyStudent = Student()

nyStudent.registrer()
```
nytt objekt

metodekall på objektet

Representasjon av verdier i Python

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

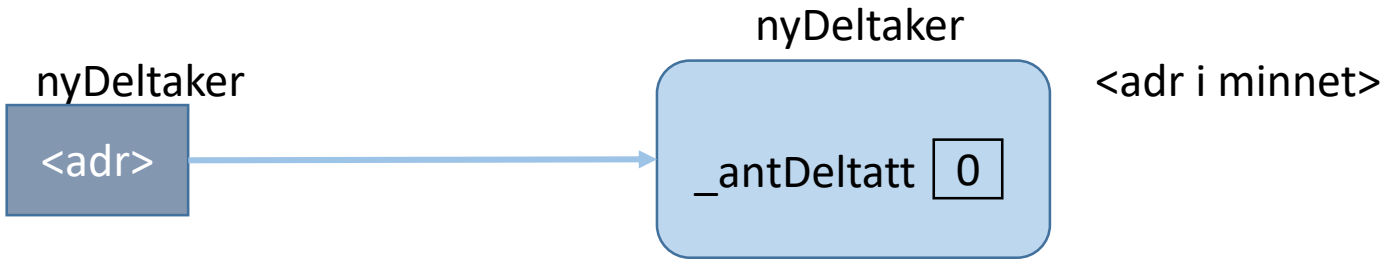
- For variabler som lagrer sammensatte verdier (objekter) trengs det et litt mer detaljert bilde:

Variablene inneholder ikke objektet selv –

men en referanse til objektet

Referanser til objekter

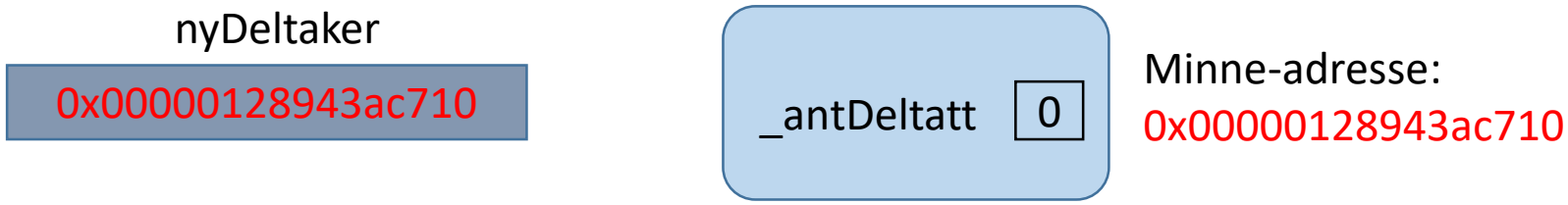
- Objekter lagres ikke direkte i variabler – variablene inneholder isteden bare referansen (adressen) til objektet.



- Tegnes ofte som en pil for å indikere at objektet ikke selv ligger i variabelen – men at variabelen kan "vise vei".
- Minneadressen – innholdet i variabelen – er en *objektreferanse* (*referanse*)

Referanser til objekter

- Litt nærmere sannheten (men veldig tungvint på tegninger...) :



=> Vi kommer til å holde oss til denne!

Referanser til objekter

- Variabler som holder rede på objekter kalles referansevariabler
- Gjør det mulig å ta vare på og bruke objekter når vi trenger dem, akkurat som heltallsvariabler husker heltall til vi trenger dem.
- Selve objektet kan lagres "hvorsomhelst" i minnet, og være stort eller lite – referansevariabelen trenger bare plass til en adresse
- Referansevariabler kan brukes for å kalle på metoder i objektet:

  refVariabel.metode()

Forskjellen på referansen og objektet

- Når endrer vi en referansevariabel og når endrer vi objektet?

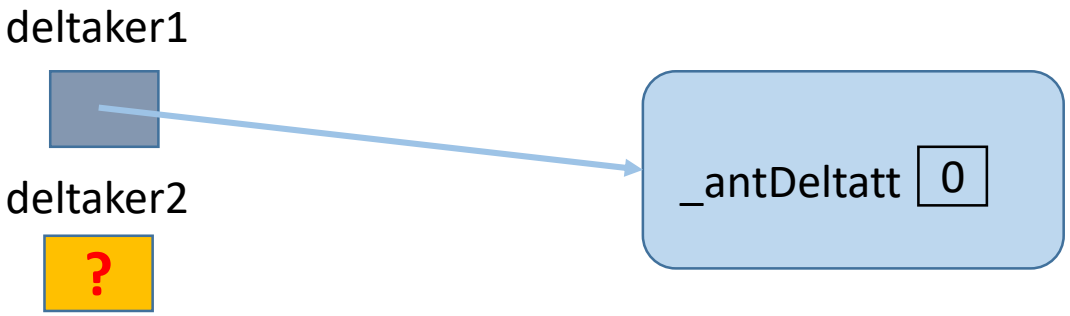
```
nyDeltaker = Student()      # Her endres innholdet i referansevariabelen
                            # (den refererer til det nye objektet)

nyDeltaker.registrer()      # 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: **`nyDeltaker.registrer()`**
  - Eller i metodene inne i klassen: **`self._antDeltatt += 1`**

Referanser til objekter

- Noen ganger er det spesielt viktig å ta hensyn til at referansen og objektet er to ulike ting!
- Hva skjer for eksempel om vi tilordner verdien fra en referansevariabel til en annen?

```
deltaker1 = Student()
deltaker2 = deltaker1
```

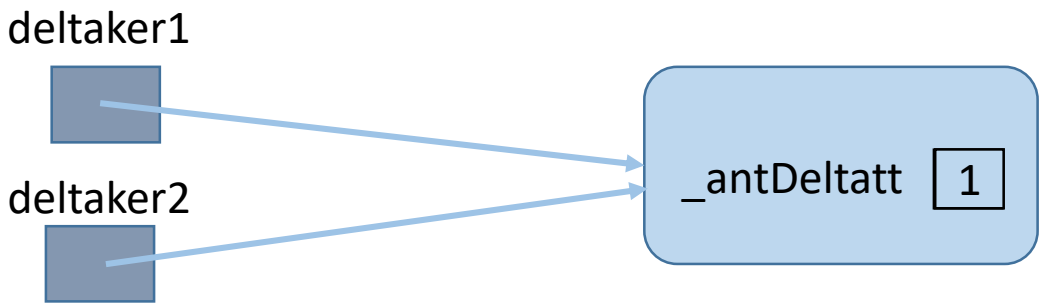


Å kopiere en referanse

- Vi kopierer *verdien* i variabelen på høyre side – akkurat som når vi kopierer en heltallsverdi
- Det er altså ikke objektet som blir kopiert – vi får bare en kopi av adressen/ referansen til objektet
- Denne verdien legges i variablen på venstre side
- Begge referansevariablene refererer dermed til *samme objekt*

Referanser til objekter

```
deltaker1 = Student()
deltaker2 = deltaker1
deltaker2.registrer()
print (deltaker1.hentDeltakelse())
```



Spørsmål:

Hva skrives ut i print-setningen?

Dette har du kanskje opplevd med lister?

```
liste = [1,2,3]
kopi = liste

kopi.append(5)

print(kopi)
print(liste)
```

- Utfør i Pythontutor

Hva skjer her? Fibonaccitall

```
def nyFib (fibTallene):
    nyttTall = fibTallene[-1] + fibTallene[-2]
    fibTallene.append(nyttTall)
    print("Liste med nytt tall: ", fibTallene)

start = [0,1]
print("Start-listen: ", start)

nyFib(start)

print("Start-listen: ", start)
```

- Fibonaccitallene: Hvert tall er summen av de to foregående
- Starter med 0, 1
- Hva skrives ut?

Hva skjer her? Fibonaccitall

```
def nyFib (fibTallene):
    nyttTall = fibTallene[-1] + fibTallene[-2]
    fibTallene.append(nyttTall)
    print("Liste med nytt tall: ", fibTallene)

start = [0,1]
print("Start-listen: ", start)

nyFib(start)

print("Start-listen: ", start)
```

- parameteren **fibTallene** får samme verdi som argumentet **start**
- den refererer med andre ord det samme liste-objektet
- append-metoden endrer liste-objektet den kalles på
- når nyFib returnerer, er listen som argumentet (start) refererte til, endret

pythontutor

Noen objekter kan ikke endres

- Noen klasser har ikke metoder som endrer instansvariablene, de kalles *immutable* – "uforanderlige"
- Hindrer feil som følge av at flere variabler refererer til samme objekt
- Strenger er eksempler på dette – alle metoder som endrer en streng returnerer et NYTT objekt i stedet for å endre det eksisterende objektet
- Må se på dokumentasjonen av klassens grensesnitt (metodene) for å vite

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

# Lister er mutable – objektet endres
liste.append("Per")
```

Strenger endres aldri – oppretter bare en ny

- Metoder for å endre en streng endrer aldri objektet de kalles på
- I stedet er de laget som funksjonsmetoder som *returnerer en ny streng*
- Denne returverdien må vi evt ta vare på for å benytte videre
- pythontutor viser streng-variabler på samme måte som tall- og sannhetsvariabler

```
navn = "Erna Solberg"

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

print(navnMedSmaaBokstaver)
print(navn)
```

Kode og tegninger

- variabler tegnes som en navngitt boks (rektangel) med verdi inni
- hvis verdien er en int, boolean eller string skriver vi den vanligvis direkte inne i boksen (selv om string er et objekt vi kan kalle metoder på)

- objekter (av mutable/ forandelige klasser) tegnes som rektangler med runde kanter
- ofte angir vi klassen (typen) over objektet – da med : foran klassenavnet
- referanser tegnes som piler fra referansevariabel til objekt
- instansvariabler tegnes inne i objektet: Som bokser eller bare <_instansvariabel> = <verdi>

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

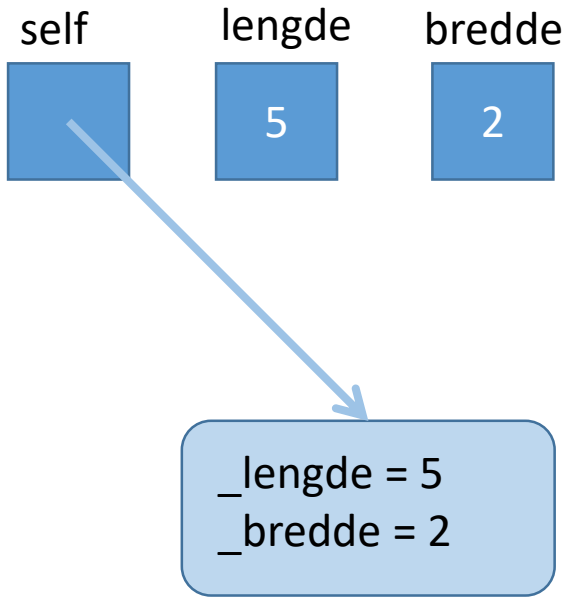
Klassen Rektangel

```
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 skjer når vi oppretter objekter?

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

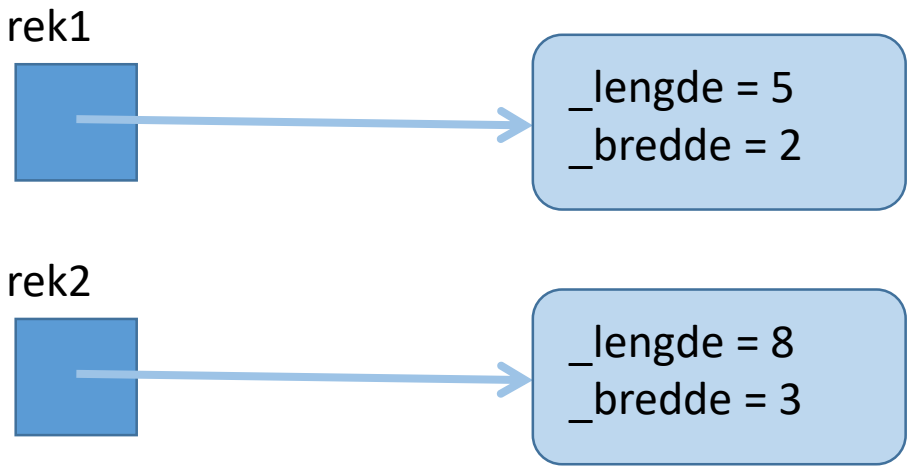
```
rek1 = Rektangel(5,2)

rek2 = Rektangel (8,3)
```



Hva skjer når vi oppretter objekter?

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

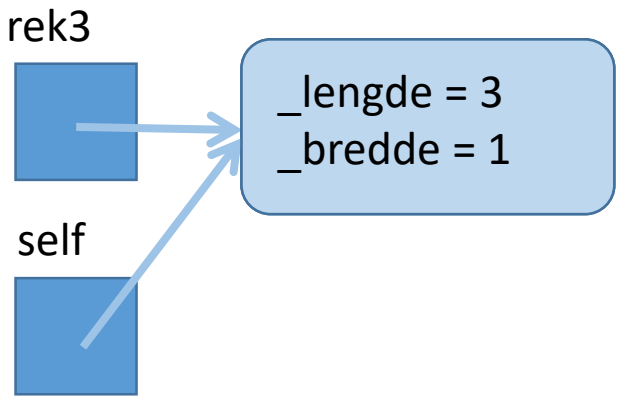
```
rek1 = Rektangel(5,2)

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

- Hører ikke til objektet slik som 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):
     areal = self._lengde*self._bredde
     return areal
```

- Parametere oppfører seg som lokale variabler – MEN initialiseres med argumentene i kallet

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 være nødvendig å sjekke mot None for å unngå å kalle på en metode i et objekt som ikke finnes (gir feil under kjøring)
- Mer om dette og sammenligning av referanser og objekter neste uke!

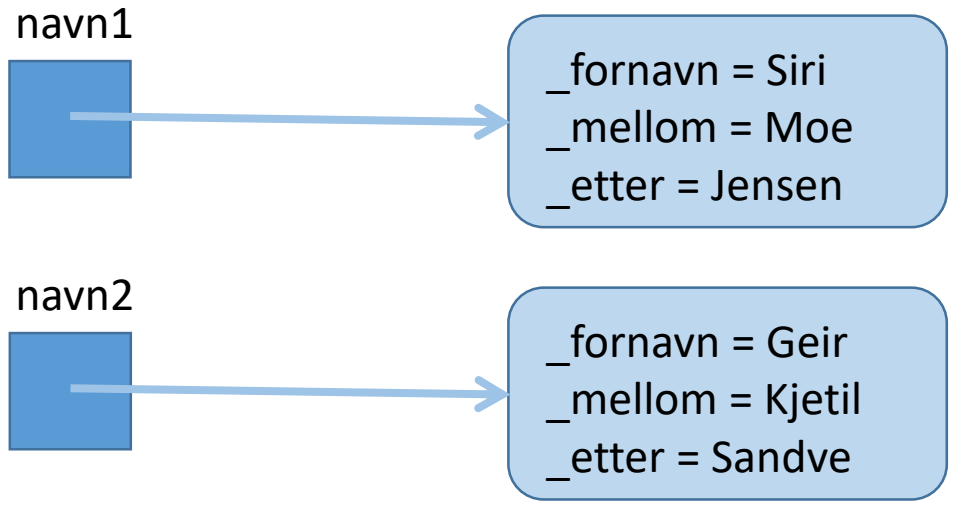
Klassen Navn fra uke 7

Filen navn.py

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

  def sortert(self):
    alfNavn = self._etter + ", " + self._fornavn + " " + self._mellom
    return alfNavn

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

Klassen Navn: Testprogram fra uke 7

```
from navn import Navn

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

print (navn1.sortert())
print (navn2.sortert())
print (navn2.naturlig())
```



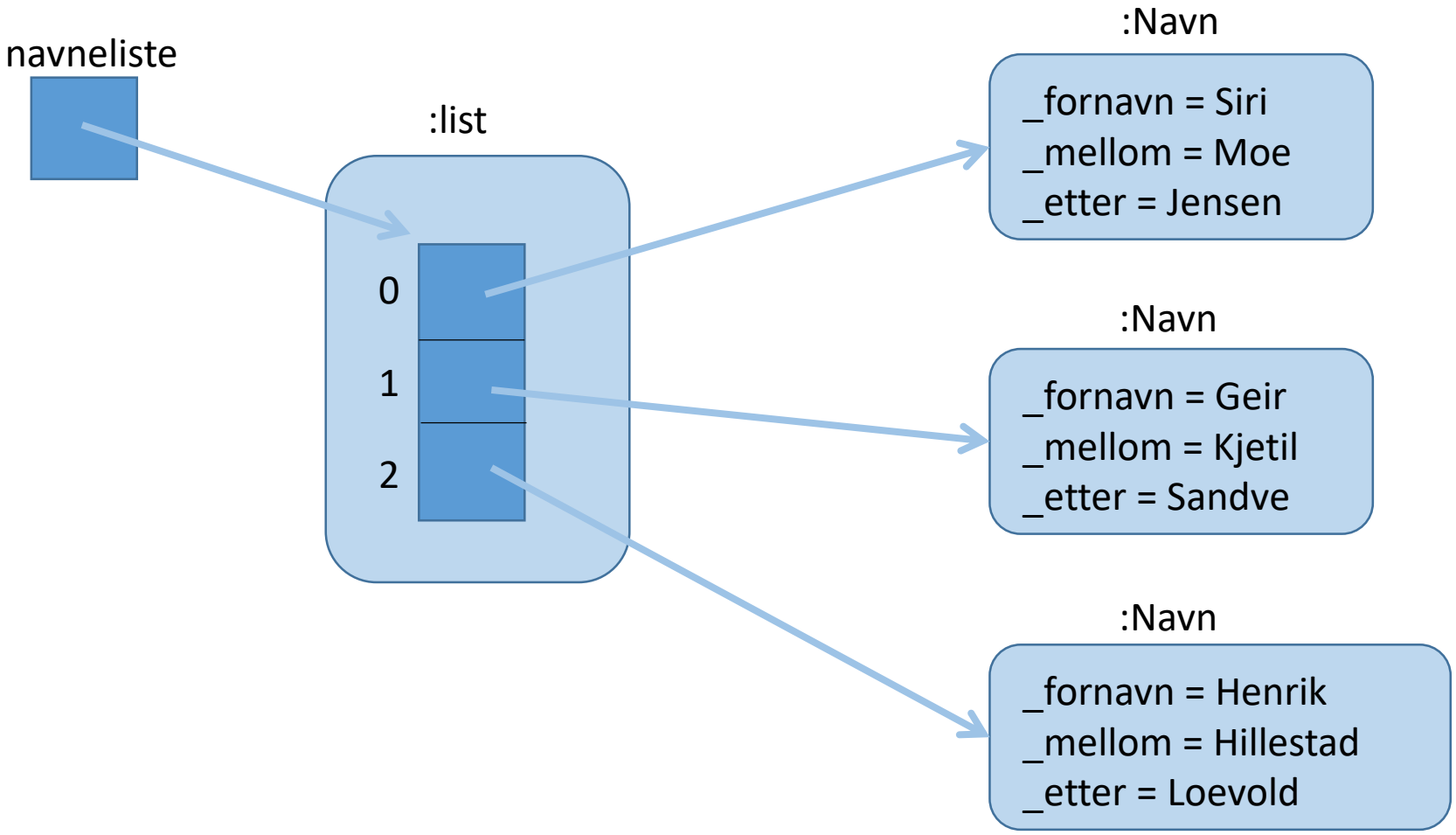

```
Jensen, Siri Moe
Sandve, Geir Kjetil
Geir Kjetil Sandve
```

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:

```
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 etNavn in navneliste:
    print(etNavn.sortert())
```

Datastrukturen etter innlesing av tre navn



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!

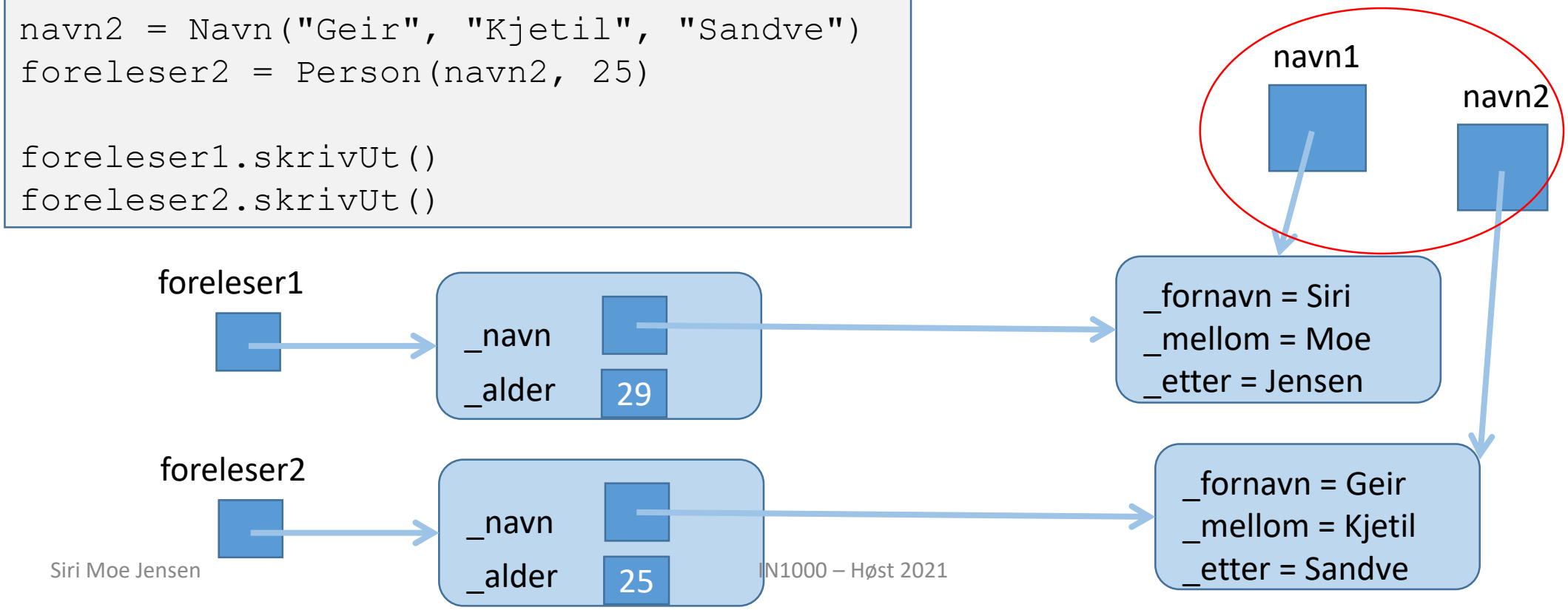
Implementasjon av Person

Filen person.py

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

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

Klassen Person: Testprogram

```
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.skrivUt()
foreleser2.skrivUt()
```



Testprogram: Flere personer

```
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: "))
    nyPerson = Person(nytt, alder)
    personliste.append(nyPerson)
    les = input("Oppgi navn på naturlig form: ")

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

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 sette seg inn i enkle programmer med flere klasser og 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