Objektorientert programmering i Python

IN1000

Høst 2026 – uke 9

Siri Moe Jensen

Obligatorisk innlevering 7

Uklar oppgavetekst?
=> Bruk Discourse
=> gjør antakelser og beskriv dem

- Hvorfor teste metodene etter hvert?
  - se klasse/ metodedefinisjon og kall på metodene i sammenheng
  - avdekke og rette feil så tidlig som mulig
  - lettere å finne ut hva som feiler og *hvorfor*
  - *Det du leverer inn skal kjøre!*
    MEN: Hvis du ikke klarer å finne/ rette en feil, beskriv i Devilry-kommentaren

- Oblig 7 tester mye av pensum

- Eksamen har vanligvis en større oppgave, med flere klasser, mer kompleksitet
  ⇒ oblig 8 programmeringen, tidligere eksamensoppgaver og *prøveeksamen*
  ⇒ Mengdetrening gir forståelse og erfaring med ulike varianter og kombinasjoner

Innhold uke 9
Mer komplekse strukturer

- Innkapsling og grensesnitt, referanser og objekter (litt repetisjon)

- Spesielle (magiske) metoder i egendefinerte klasser
  - sammenligne et objekt med et annet objekt
  - lage en tekststreng av innholdet i objektet

- Samlinger og datastrukturer, flere eksempler og mer kompleksitet
  - objekter i beholdere (containers) som liste, mengde og ordbok.
  - aggregering: Klasser som bygger på andre klasser

Klasse og grensesnitt

- Klassen er mønsteret vi lager objektene etter
- Klassedefinisjonen bestemmer hva objektene kan lagre av data og hvordan (i instansvariabler) og hva de kan utføre (metoder)
- Grensesnittet er de metodene vi tilbyr programmerere som skal bruke klassen
- Alt annet i klassen navngir vi med _ først i navnet, de er *non-public*. Dette gjelder alle instansvariabler, dvs hvordan vi representerer dataene våre.
- Metoder kan også være non-public! Brukes da kun av andre metoder inne i klassen, og får navn som starter med _

Klassen Rektangel fra uke 8

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

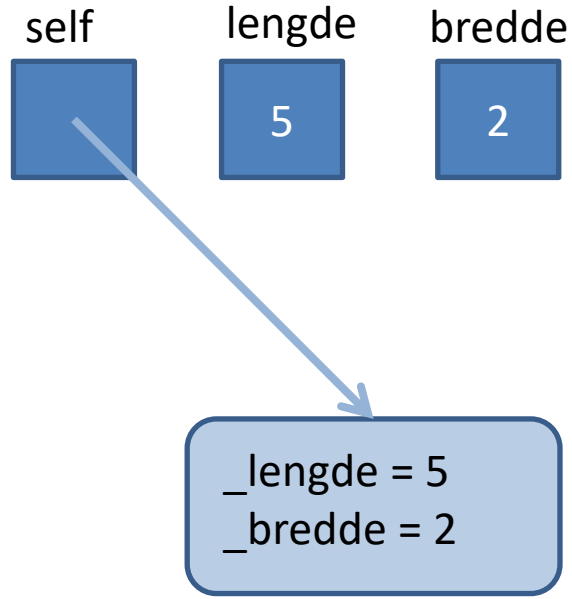
  def areal(self):
    areal = self._lengde*self._bredde
    return areal

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

Klassen Rektangel fra uke 8 – opprette objekt

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

rek1 = Rektangel(5, 2)
```



Hva kan vi bruke når vi skriver en metode? I

Data:

- **instansvariablene** (ved hjelp av .self). Når en metode avslutter, finnes fortsatt instansvariablene med sine verdier (så lenge det finnes en eller flere referanser til objektet)

Dessuten (akurat som vanlige funksjoner og prosedyrer)

- eventuelle **parametere** til metoden (eks: lengde, bredde). Disse oppstår når metoden blir kalt, og får verdi fra argumentene som sendes med. Når metoden er utført forsvinner de.
- eventuelle **lokale variabler** metoden definerer (eks: mitt_areal). Som parametere, men må gis verdi inne i metoden.

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

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

Hva kan vi bruke når vi skriver en metode? II

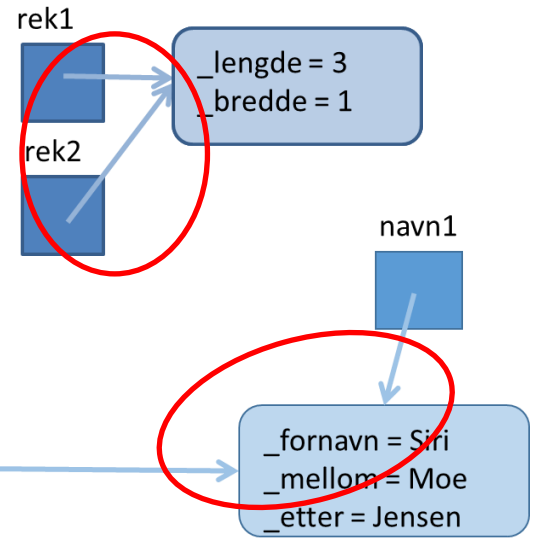
Prosedyrer/ funksjoner/ metoder:

- En metode kan bruke (kalle på) **funksjoner og metoder** akkurat som ellers i programmet:
  - funksjoner: print("tekst") eller input("svar: ")
  - andre metoder: referanse.metode(argumenter)

Referanse kan være

- til det objektet metoden ble kalt på:
  self._naturlig() i klassen Navn
- til et annet objekt
  self._navn.naturlig() fra klassen Person

Referanser til objekter



- Spesielt viktig å huske forskjellen på referansen og objektet

1. hvis vi tilordner verdien fra en referansevariabel til en annen
   - Innholdet (verdien) i variabelen kopieres
     innholdet i referansevariabelen er en adresse
   - *objektet* kopieres ikke
   - variablene refererer til samme objekt!

   => hvis objektet endres vises det gjennom begge referansevariabler

2. parameter-overføring er eksempel på kopiering av verdi fra argument til parameter
   (obs: gjelder også parametere i konstruktører)

3. ved sammenligning (samme objekt dvs like referanser, eller like objekter?)

Finn problemer med koden (programmet virker)

```
def skriv_deltakere(liste, max_ant):
    while len(liste) > max_ant:
        liste.pop()
    print("Disse fikk plass: ", liste)

def hovedprogram():
    kø = ["Per", "Pål", "Espen"]
    print("Kø:", kø)
    skriv_deltakere(kø, 2)
    print("Kø: ", kø)

hovedprogram()
```

Skriv et program som gitt en kø og et maks antall deltakere, skriver ut hvilke deltakere som får plass.

```
Kø: ['Per', 'Pål', 'Espen']
Disse fikk plass:  ['Per', 'Pål']
```

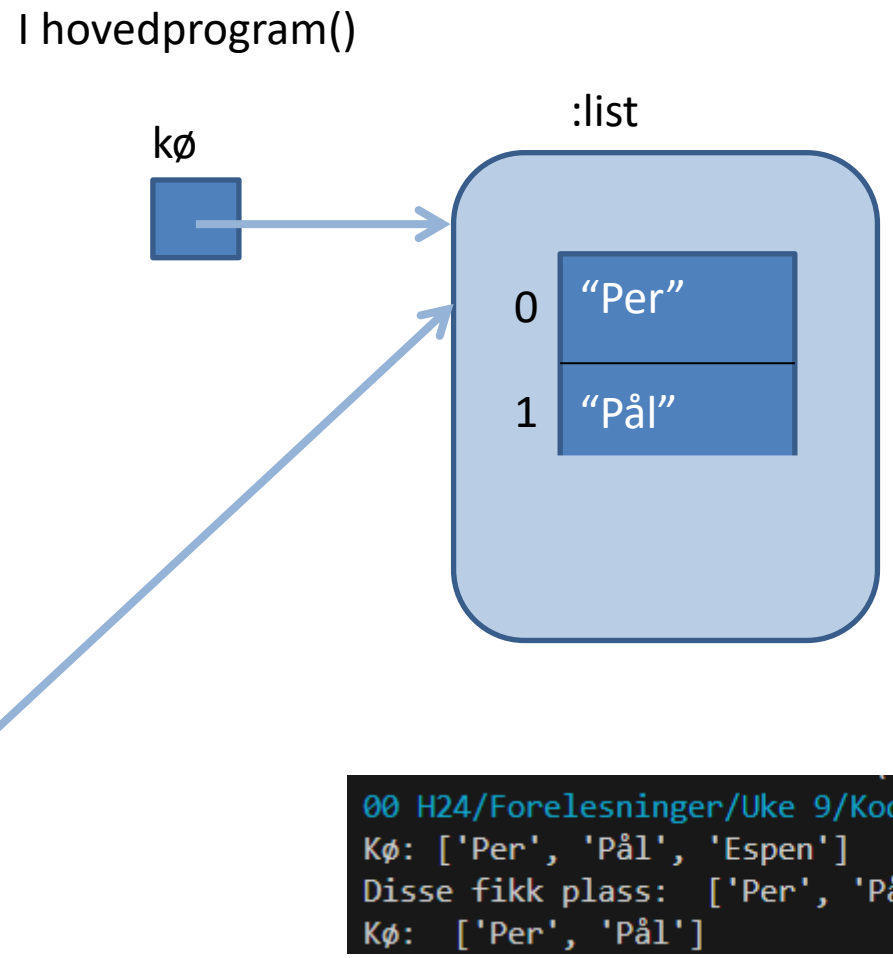
- Funksjonen har to svært ulike oppgaver (kø-adm og utskrift)
- Den endrer parameteren **liste** (noe som får effekt i hovedprogrammet).

Obs: Parameter-overføring

```python
def skriv_deltakere(liste, max_ant):
    while len(liste) > max_ant:
        liste.pop()
    print("Disse fikk plass: ", liste)

def hovedprogram():
    kø = ["Per", "Pål", "Espen"]
    print("Kø:", kø)
    skriv_deltakere(kø, 2)
    print("Kø: ", kø)

hovedprogram()
```



[I Pythontutor]

```
00 H24/Forelesninger/Uke 9/Kode/deltakere.py"
Kø: ['Per', 'Pål', 'Espen']
Disse fikk plass:  ['Per', 'Pål']
Kø:  ['Per', 'Pål']
```

Lister er mutable/ muterbare

- Mange liste-metoder endrer listen (objektet), f eks:
  pop, append, remove og oppdatering via indeks (`li[0] = 5`)

```
li = [1,2,3]
li2 = li
li[0] = 5
print(li2)
```

```
>>> li = [1,2,3]
>>> li2 = li
>>> li[0] = 5
>>> print(li2)
[5, 2, 3]
>>>
```

- Krever forsiktighet ved kopiering av liste-variabler, inkludert parameteroverføring
- Gjelder generelt for muterbare klasser, også de dere skriver selv
- To referanser til samme objekt gjør at endringer i objektet har effekt for begge

- Ofte nyttig, men krever oppmerksomhet: Hva er det mest logiske/ konsistente, hvordan navngi, *dokumenter*.

Kø-eksempelet

- "side-effekt" av funksjonen **`skriv-deltakere`** er at den endrer listen
  Navnet antyder at funksjonen kun gjør en utskrift, mens den faktisk også velger deltakere, *og* endrer køen

- Ryddigere: Returner deltakerliste i stedet for å skrive ut, gjør et valg om køen skal modifiseres eller ikke, eventuell endring av listen bør være meningsfylt, velg navn som beskriver funksjon, dokumenter effekt

Kunne det vært fornuftig å endre en køen i funksjonen?

Ja, men fortrinnsvis da beholde de som ikke fikk plass – og helst i en ny funksjon som *kun* flytter fra kø til en deltakerliste (som returneres). Eks:

{flytt_fra_kø.py}

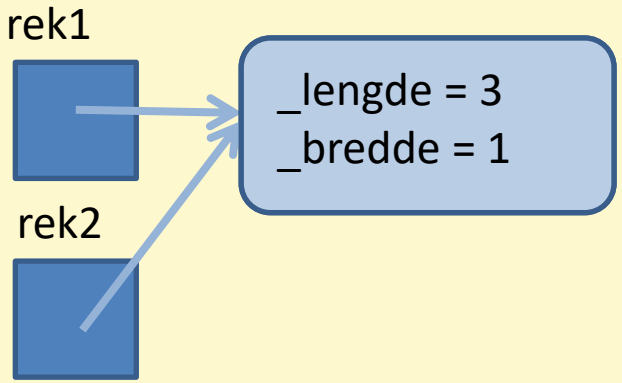
```
def ta_ut_deltakere(kø, max_ant):
    deltakere = []
    while len(deltakere) < max_ant and len(kø) > 0:
        deltakere.append(kø.pop(0))
    return deltakere
```

Sammenligning av referansevariabler

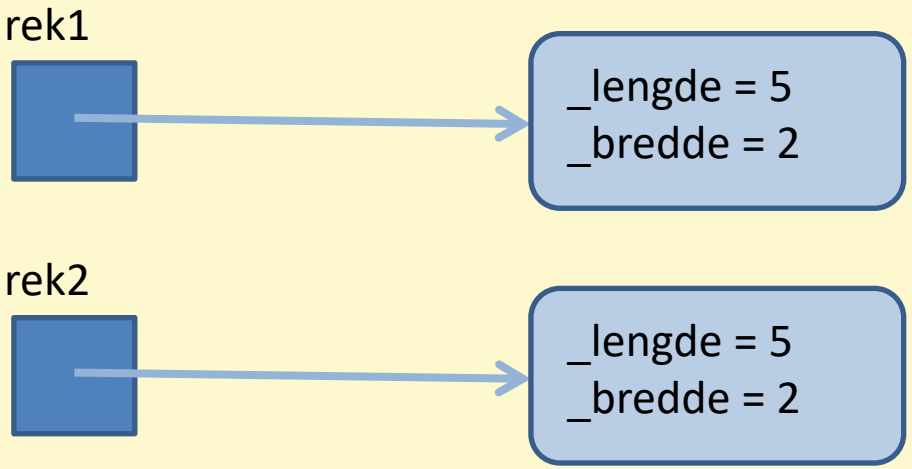
Kan sammenligne enten:

- Referansene: Er det samme objekt (lik adresse)?
  - Bruk `is` for å sjekke

    `rek1 is rek2` *True*



- Objektene: Er det like objekter (likt innhold)?
  - Må sjekke instansvariablene i begge objekter!
  - `rek1 is rek2` *False*
  - `rek1 == rek2` *????*



Når er *objekter* like?

- Den som har skrevet klassen vet (bestemmer) hva som gjør to objekter "like".

- Dette er "ivaretatt" for de innebygde typene dere har brukt hittil (int, string, list..)
  - For eksempel for lister, slik at

    `liste1 == liste2`

    sammenligner hvert element i liste1 og liste2 for oss

- Hvis det ikke er bestemt i klassen hva som avgjør likhet, sjekker Python bare om det er samme objekt (lik adresse), dvs True hvis a is b, ellers False

- Vi kan lage en egen metode i våre klasser for sammenligning. Hvis metoden får navnet `__eq__` vil operatoren `==` sammenligne to objekter av klassen slik vi bestemmer.

```
if rek1 == rek2:
  print("Like!")
```

eksempel på __eq__ metode

- Vil gjøre det mulig å sammenligne to rektangler for likhet
- Vi bestemmer at to Rektangel-objekter er like hvis og bare hvis lengden er den samme i begge, og bredden er den samme i begge
- Trenger en __eq__ metode i klassen som skal sammenligne instansvariablene for lengde og bredde i self-objektet, med instansvariablene i et annet objekt
- __eq__ metoden tar som parameter (i tillegg til self) en referanse til et annet objekt (som vi skal sammenligne med) og returnerer True eller False
- Argumenter til disse parametrene er referansene (objektene) på hver side av **==**

eksempel på __eq__ metode

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

  def __eq__(self, annen):
    return (self._lengde == annen._lengde and
            self._bredde == annen._bredde)

r1 = Rektangel(8,6)
r2 = Rektangel(8,6)
print (r1 == r2)

r3 = Rektangel(6,4)
print (r2 == r3)
```

True

False

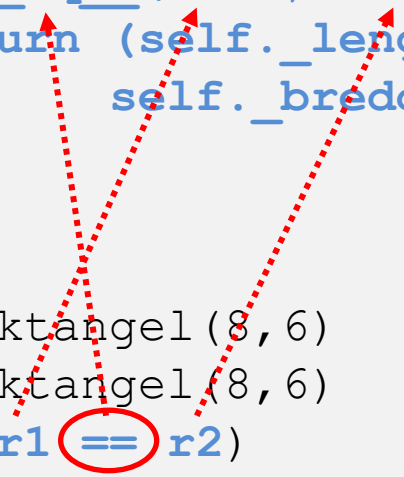
Når og hvordan kalles __eq__metoden?

```
class Rektangel :
  # konstruktør

  def __eq__(self, annen):
    return (self._lengde == annen._lengde and
            self._bredde == annen._bredde)



r1 = Rektangel(8,6)
r2 = Rektangel(8,6)
print (r1 == r2)
```



Operatoren **==** gjør at

- Python kaller **__eq__** på referansen på venstre side
- Argumentene til **__eq__** blir referansene på hver side av **==**
- Hvis klassen til referansen på venstre side ikke har egen **__eq__**, testes det bare om de refererer samme objekt (**True**), ellers **False**

En bedre __eq__ metode

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

  def __eq__(self, annen):
    if type(annen) is not Rektangel:
      return False
    return (self._lengde == annen._lengde and
            self._bredde == annen._bredde)

r1 = Rektangel(8,6)
r2 = Rektangel(8,6)
print (r1 == r2)

r3 = Rektangel(6,4)
print (r2 == r3)
```

Spørsmål:

1. Er dette et brudd på innkapslingen i klassen Rektangel?

NEI – så lenge **annen** er et Rektangel- objekt

2. Ser du noen mulighet for at dette kan gi en feil under kjøring?

JA – hvis **annen** ikke refererer et Rektangel.

Sammenligning av *objekter*

```
print (o1 == o2)
```

Overloading/ overlasting av operatoren ==

Hvis referansene er like (refererer samme objekt) => **True**

Hvis de refererer hvert sitt objekt av en klasse som ikke har **__eq__** metode:
=> **False**

(selv om instansvariablene i objektene har nøyaktig like verdier, Python vet ikke hva "lik" betyr)

Hvis det finnes en **__eq__** metode i klassen:

- **__eq__**metoden utføres for o1, med o2 som argument => tilsvarer "o1.__eq__(o2)"
- Avhengig av hvordan **__eq__ metoden** er implementert vil den returnere **True** eller **False**

=> Uttrykket (o1 == o2) evaluerer til returverdien fra **__eq__** metoden i klassen til o1 og o2

Spesielle (magiske) metoder

- `__eq__` er en av flere "magiske" metoder som har en spesiell betydning i Python
- `__init__` har dere allerede brukt
- felles er
  - innledende og avsluttende dobbel underscore (__) i metodenavnet (dunder-metoder!)
  - kalles på andre måter enn ved metodenavnet
  - eks:
    - `__eq__` for `==` mellom to referanser
    - `__init__` ved opprettelse av nytt objekt
    - `__str__` kalles når vi bruker str(a) eller print(a)

Flere spesielle metoder

- Tabell 9.1 i boka viser en rekke andre spesielle metoder som kan *overlaste* (overloade) logiske (eks `==`, `!=`, `<`) og aritmetiske (eks `+`, `*`) operatorer for en klasse som trenger det

- `__init__` kalles ved opprettelse av nytt objekt: Oppretter og initierer (gir startverdi til) instansvariablene og returnerer referanse til det nye objektet

- `__str__` og `__repr__` returnerer innholdet i et objekt som en string

Objekter som strenger

For å vise frem objekter (som strenger):

- `__str__`
  - kalles når vi bruker `print(referanse)` og `str(referanse)`
  - lager *brukervennlig* utskrift, lag den slik du ønsker
  - hvis den ikke finnes i klassen kalles `__repr__`

Lag gjerne denne

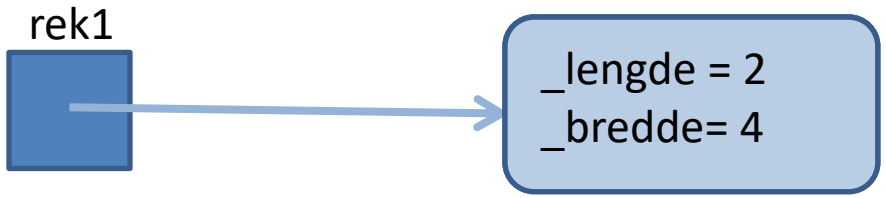
Python dokumentasjonen:
..the "informal" or nicely printable string representation of object..

- `__repr__`
  - leverer en *komplett og entydig* representasjon av objektet, mest nyttig for utvikler
  - vanlig default: Modul , klassenavn og minneadresse/ entydig id for objektet
    for tall, boolske verdier og strenger: verdien selv
  - kalles som repr() eller når `__str__` ikke finnes i klassen
  - for elementer ved print() på lister og ordbøker

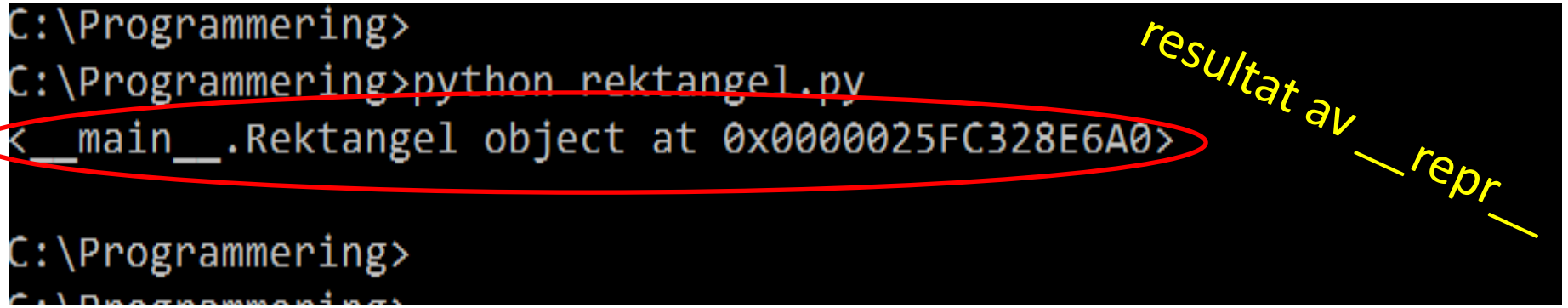
Finnes fra før, ikke rør

"Skrive ut" objekter

Printer vi en referanse til en egen klasse, får vi en (litt kryptisk) tekst ut:




```
print(rek1)
```



eksempel på __str__ metode

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

 def __str__(self) :
    pen_streng = "Lengde: " + str(self._lengde) + \
        ", bredde: " + str(self._bredde)
    return pen_streng

r1 = Rektangel(8,6)
print (r1)
```

```
>python rek.py
Lengde: 8, bredde: 6
```

⇒Husk at en __str__ metode skal returnere, ikke printe ut en streng!
(Du vet ikke hva de som bruker klassen ønsker å gjøre med strengen)

Litt mer om repr() og str()

- print kaller __str__ på argumentet hvis __str__ er implementer, ellers _repr__

- Men om du printer en liste eller ordbok, kalles repr() for elementene
  - For enkle typer (int, float, boolean, string) returnerer repr() verdien, så det ser ok ut
  - For egendefinerte objekter, kall print() eller str() på hvert enkelt element

- Husk: om du konkatinerer et objekt med en string, må du selv kalle __str__ på objektet først
  (dette er du vant til med f eks heltall)

Referanser er verdier

- En referanse kan ligge i en variabel (som rek1) – eller være en returverdi fra en funksjon eller metode (som igjen kan være kalt på en referanse som var returverdi fra en metode – osv osv)

- Et uttrykk som *evaluerer til en referanse* kan brukes der en referansevariabel kan brukes (akkurat som vi har sett med uttrykk av andre typer, for eksempel heltall)

```
from rektangel import Rektangel

                                    # Oppretter nytt Rektangel objekt og
print(Rektangel(10,15).areal())     # kaller på metoden areal for dette,
                                    # printer til slutt returverdi fra areal
```



Spørsmål:
- Kan vi etter denne linjen endre lengde eller bredde på rektangelet vi opprettet?

Vi kan bruke "dot-notasjon" i flere ledd

- Kaller da metoder på returverdi fra en annen metode

```
ordliste = "Dette er en tekst  ".lower().strip().split()
print(ordliste)
```

Spørsmål:
- Kunne vi byttet rekkefølgen på de to siste?
- Hva skrives ut slik det står her?

- Svar:
  - nei, split() returnerer (referanse til) en liste, ikke en string (som strip() må kalles på)
  - ['dette', 'er', 'en', 'tekst']

- NB: Rekkefølgen er ikke likegyldig, utføres fra venstre mot høyre!

- Tenk lesbarhet – mellomlagring i variable (med gode navn) kan gjøre koden mer lesbar (ikke minst for deg selv – tvinger deg til å tenke rekkefølge og typer)

Samlinger av verdier (se uke 3)

- Beholdere (containers) er viktige verktøy i programmering,
- Gjør det mulig å organisere og arbeide med samlinger av objekter
- Beholdere tilbyr ulike egenskaper – velges ut fra behov

- Så langt har vi sett på flere beholdere som er innebygget i Python:
  - Lister (list). Inneholder elementer med fast rekkefølge, nummerert
  - Mengder (set). Unummerert, uten dubletter
  - Ordbøker (dict). Par av unik *nøkkel* (typisk tekst, tall) og *verdi.* Ingen konstant rekkefølge*

- Verdiene kan være referanser til
  - objekter av egendefinerte klasser (Person, Navn, Rektangel, ..)
  - andre samlinger som lister, mengder eller ordbøker

* I eldre Python-versjoner

Eksempel: Informatikk-emner (kurs)

- Vi skal lage et program for å velge informatikk-emner. Initielle krav:
  - Lagre id (string emnekode), antall poeng og høst eller vår-semester (string "høst" eller "vår") for alle emner
  - Lese disse dataene fra terminal eller fil
  - Skrive ut informasjon om alle emner

```
IN1000(høst): 20 studiepoeng
IN1010(vår): 20 studiepoeng
```

- Designer en klasse **`Emne`** med instansvariabler som over, og en __str__ metode
  - Velger å initialisere instansvariablene fra parametere til konstruktøren

- Bruker en liste for å organisere **`Emne`**-objektene

En klasse for emner

```
class Emne:
    def __init__ (self,emnekode,sem,stp):
        self._emnekode = emnekode
        self._semester = sem
        self._studiepoeng = stp

    def __str__ (self):
        s = (self._emnekode + "(" +
                self._semester + "): " +
                str(self._studiepoeng) + " studiepoeng")
        return s
```

Samlinger av Emne-objekter

- Mer presist: Samlinger av referanser til Emne-objekter
- En liste kan inneholde (referanser til) objekter
- Dictionaries (ordbøker) kan også ha referanser som *verdier*, typisk med en (unik) instansvariabel fra objektet som nøkkel
- Eks: Dictionary med Emne-objekter
  - Nøkkel: Emnekode
  - Verdi: Referanse til objektet for det emnet

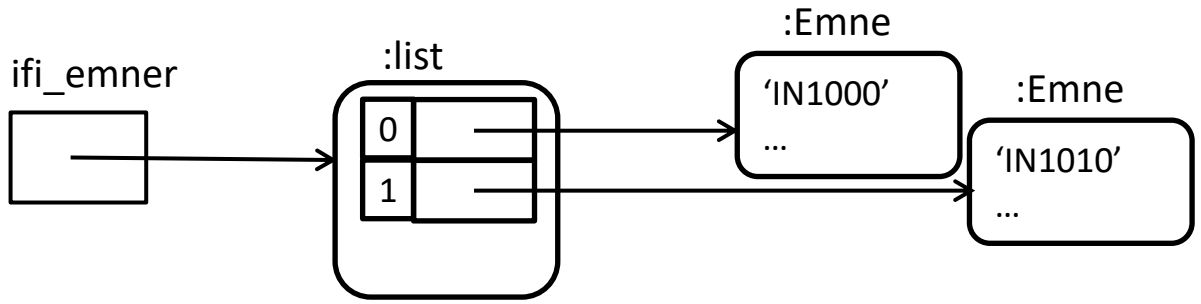
Liste med objekter av egen klasse

- Eksempel: ifi_emner
- Hvert element i listen er (en referanse til) et emne-objekt

```
from emne import Emne

ifi_emner = []
ifi_emner.append(Emne("IN1000","høst",20))
ifi_emner.append(Emne("IN1010","vår",20))  # ..osv osv

for ett_emne in ifi_emner:
    print(ett_emne)                  # kaller __str__()
```




```
IN1000(høst): 20 studiepoeng
IN1010(vår): 20 studiepoeng
```

test_emne_liste.py > ...

```python
from emne import Emne

ifi_emner = []
ifi_emner.append(Emne("IN1000","høst",20))
ifi_emner.append(Emne("IN1010","vår",20))  # ..osv osv

for ett_emne in ifi_emner :
    print(ett_emne)                 # kaller __str__()

print(ifi_emner)
```

TERMINAL PORTS PROBLEMS OUTPUT DEBUG CONSOLE Python

```
r/Uke 9/Kode/test_emne_liste.py"
IN1000(høst): 20 studiepoeng
IN1010(vår): 20 studiepoeng
[<emne.Emne object at 0x000001AFA5D6E310>, <emne.Emne object at 0x000001AFA5D6DF50>]
```

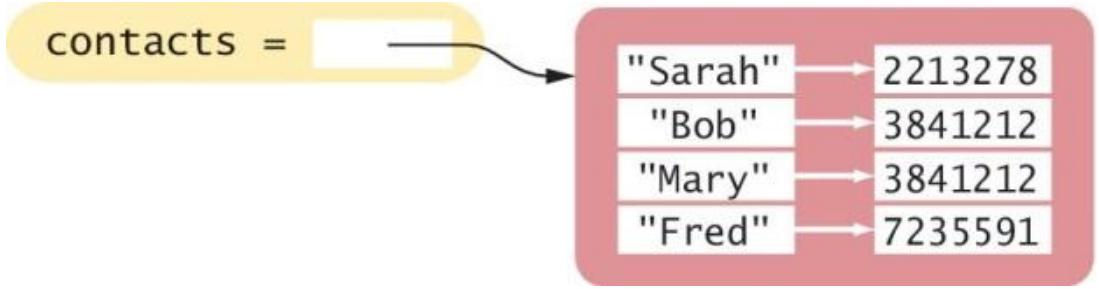
Opprette Dictionary (ordbok)

(illustrasjon fra *Python for Everyone*)

- Et program som slår opp telefonnummer

- Bruker en ordbok der navn er nøkkel, og telefonnummer er verdien

```
contacts = { "Fred": 7235591, "Mary": 3841212, "Bob":
   3841212, "Sarah": 2213278 }
```



Ordbok med referanser

- Verdiene i en ordbok (dict) kan være referanser til objekter

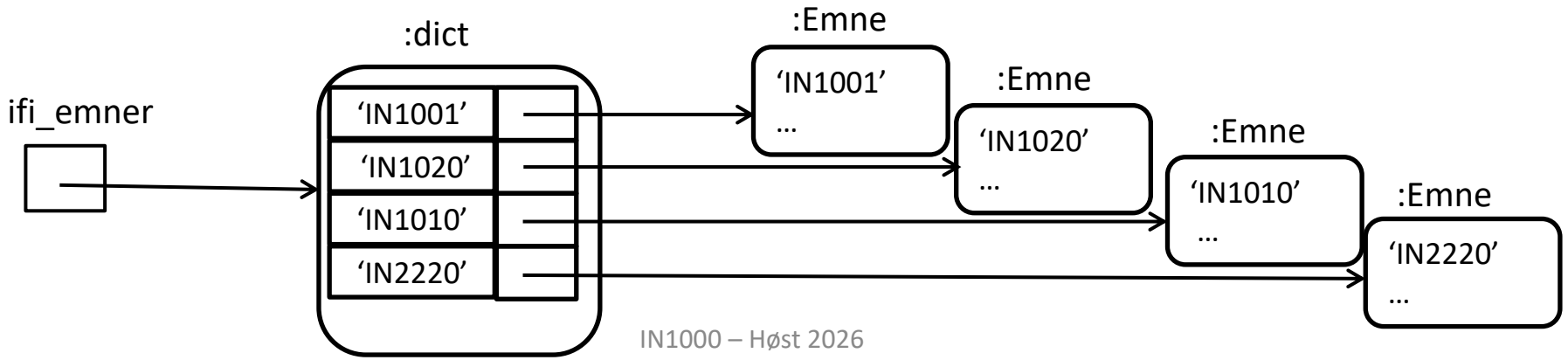
- Eksempel: Ordbok `ifi_emner` med emner
- Nøkkel (entydig streng): Emnekode
- Verdi: Referanse til et Emne-objekt

Ordbok med referanser II

```
from emne import Emne

ifi_emner = {}              #opprettet tom ordbok

ifi_emner["IN1000"] = Emne("IN1000","høst",10)
ifi_emner["IN1010"] = Emne("IN1010","vår",10)
# etc
```



Hvordan løpe gjennom (iterere over) en ordbok?

```python
from emne import Emne

ifi_emner = {}              #opprettet tom ordbok

ifi_emner["IN1000"] = Emne("IN1000","høst",10)
ifi_emner["IN1010"] = Emne("IN1010","vår",10)
# etc

for ett_emne in ifi_emner.values():
    print(ett_emne)     # Kaller __str__()

print()
for emnekode in ifi_emner:
    print(emnekode)

print()
for emnekode in ifi_emner:
    print(ifi_emner[emnekode])
```

Hva skrives ut i løkkene?

```
IN1000(høst): 10 studiepoeng
IN1010(vår): 10 studiepoeng

IN1000
IN1010

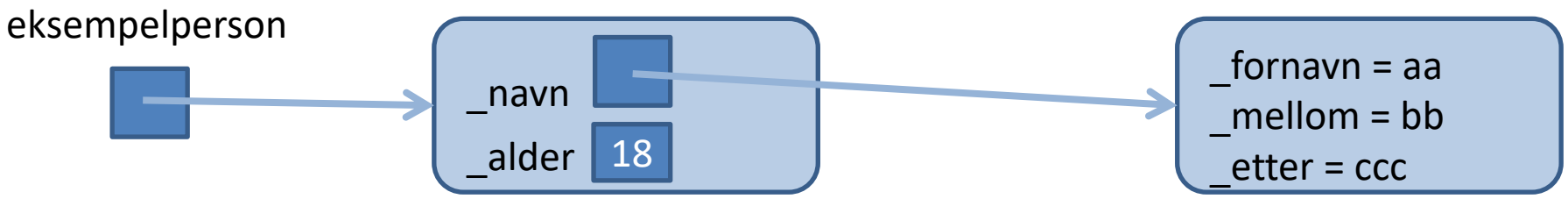
IN1000(høst): 10 studiepoeng
IN1010(vår): 10 studiepoeng
PS M:\Ifi\Undervisning\IN1000 H
```

Implementasjon av Person

Filen person.py

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

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



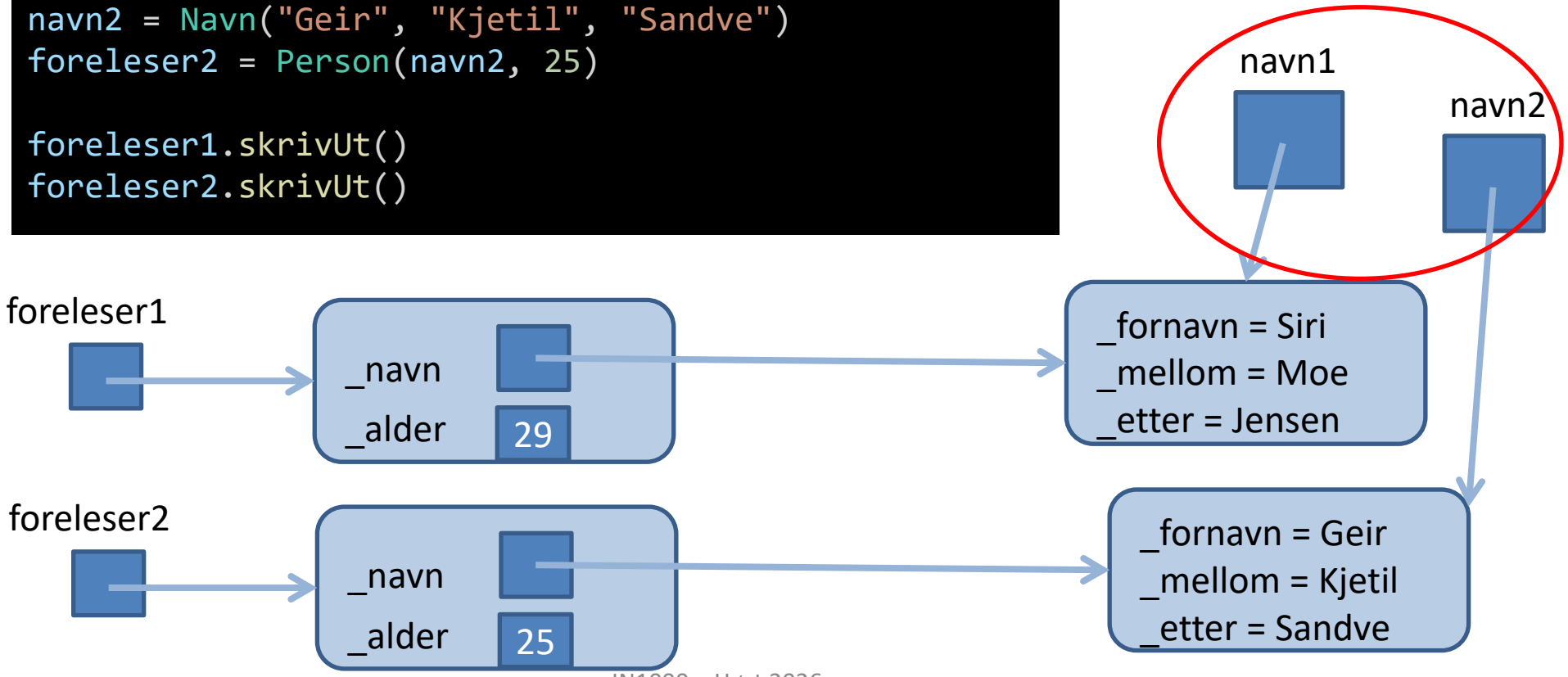
Klassen Person: Testprogram

```
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 I: Bygge struktur

```
from navn import Navn
from person import Person

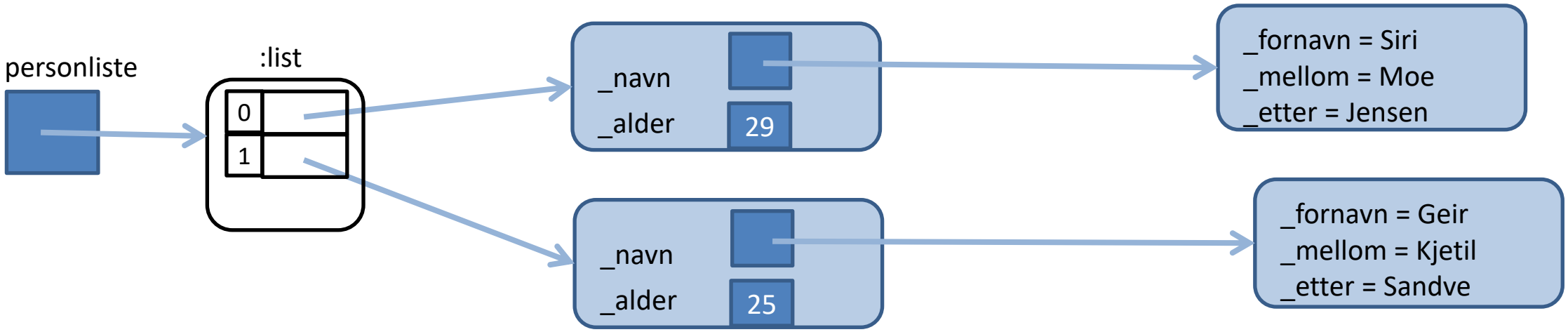
personliste = []
les = input("Oppgi navn på naturlig form: ")
while les != "" :
    navnene = les.split()
    nytt = Navn(navnene[0],navnene[1],navnene[2])
    alder = int(input("Oppgi alder: "))
    nyPerson = Person(nytt, alder)
    personliste.append(nyPerson)
    les = input("Oppgi navn på naturlig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

for person in personliste :
    person.skrivUt()
    print()
```

Testprogram I: Bygge struktur

```
from navn import Navn
from person import Person

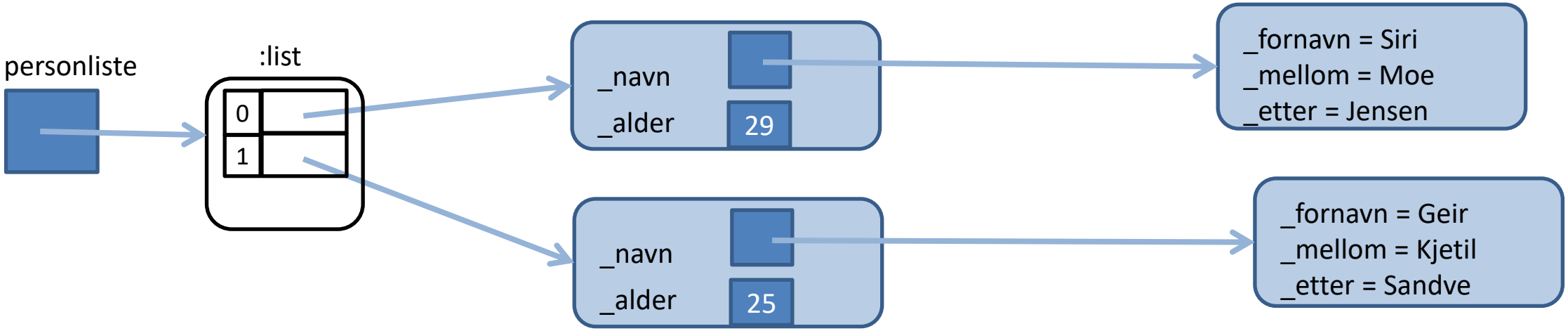
personliste = []
les = input("Oppgi navn på naturlig form: ")
while les != "" :
    navnene = les.split()
    nytt = Navn(navnene[0],navnene[1],navnene[2])
    alder = int(input("Oppgi alder: "))
    nyPerson = Person(nytt, alder)
    personliste.append(nyPerson)
    les = input("Oppgi navn på naturlig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



Testprogram II: Skrive ut alle personer

```
<som i forrige lysark>
:
while les != "" :
    :
    <som i forrige lysark, legg Person-objekter i personliste>

for person in personliste :
    person.skrivUt()
    print()
```



Person og Navn: Eksempel på aggregering

- Person-klassen bruker en annen klasse for å representere (noen av) sine data
- Gir en avhengighet: Person-klassen virker ikke uten Navn-klassen (eller om grensesnittet til Navn endrer seg)

=> Person er en *aggregert klasse*

- Merk at Navn-klassen her kan brukes uavhengig av Person-klassen

- Vi kan aggregere flere klasser. For eksempel lage en klasse Gruppe, som har et nummer og ingen eller flere personer.

Oppgave: Skriv metoden __str__ i klassen Navn

```
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

  def __str__(self):
    return self.naturlig()
```

Oppgave: Skriv metoden __str__ i klassen Person

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

  def __str__(self):
    s = "Navn: " + str(self._navn) + ". Alder: " + str(self._alder)
    return s
```

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

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

print(foreleser1)
print(foreleser2)
```

```
PS M:\Ifi\Undervisning\IN1000 H26\Forelesninger\Uke 9\Kode publ> python .\test_person.py
Navn: Siri Moe Jensen. Alder: 29
Navn: Geir Kjetil Sandve. Alder: 25
```

En klasse med en samling som instansvariabel

- Kan få en ferdig liste (eller ordbok) i konstruktør-parameter
- eller lage en tom liste (ordbok) i konstruktøren
  og ha en metode som legger til et nytt elementer i samlingen

- Ny klasse Gruppe med instansvariabler
  - gruppenummer
  - en liste med gruppemedlemmer (Person-referanser)

En klasse Gruppe med Personer med Navn

```python
class Gruppe:
    def __init__(self, nr):
        self._gruppe_nr = nr
        self._medlemmer = []

    def nytt_medlem(self, ny):
        self._medlemmer.append(ny)

    def hent_alle_medlemmer(self):
        return self._medlemmer
```

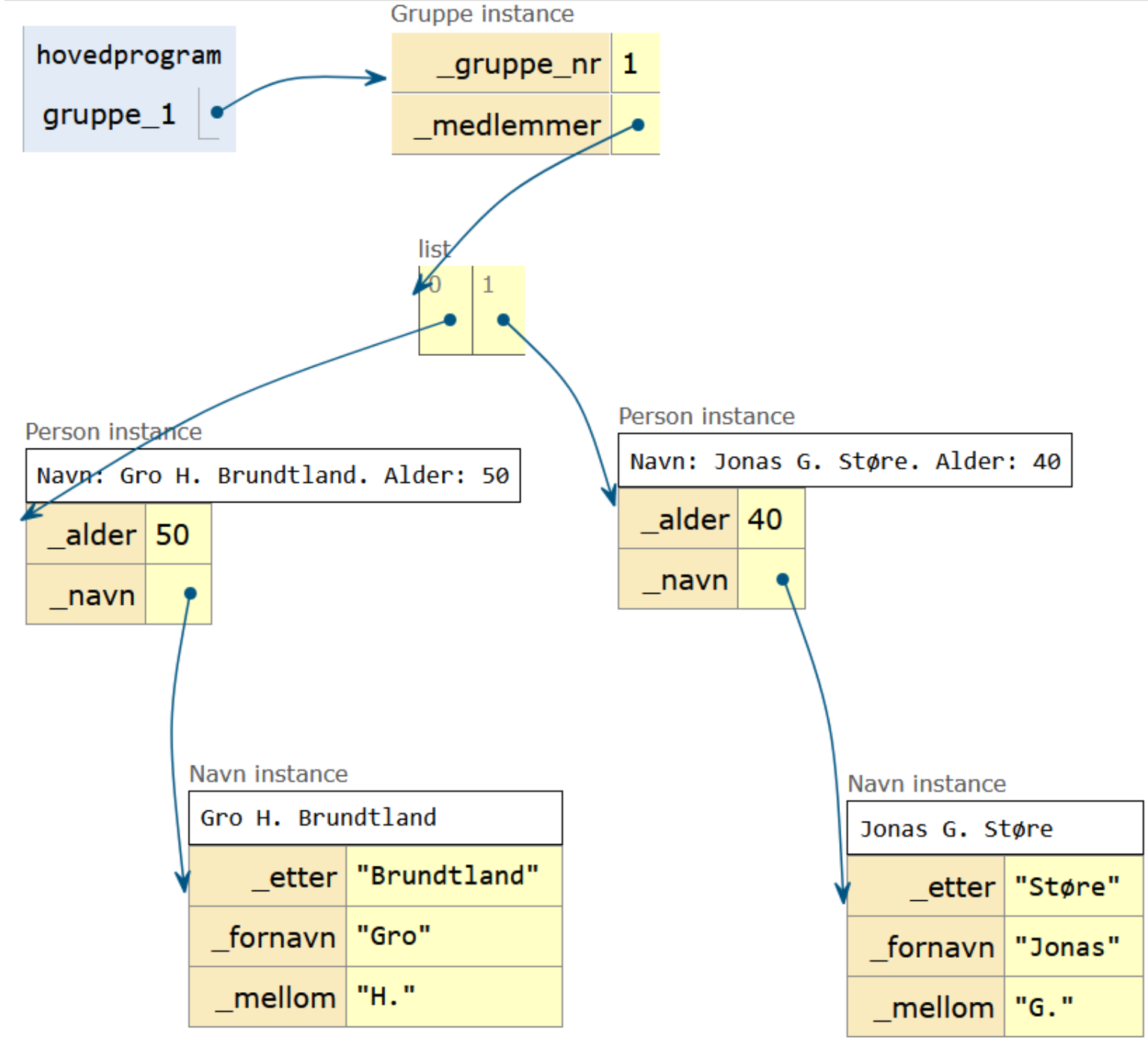
```python
from navn import Navn
from person import Person
from gruppe import Gruppe

def hovedprogram():
    gruppe_1 = Gruppe(1)
    ny_person = Person(Navn("Gro", "H.", "Brundtland"), 50)
    gruppe_1.nytt_medlem(ny_person)
    ny_person = Person(Navn("Jonas", "G.", "Støre"), 40)
    gruppe_1.nytt_medlem(ny_person)
    medlemmer = gruppe_1.hent_alle_medlemmer()
    for medlem in medlemmer:
        print(medlem)

hovedprogram()
```

```
00 H24/Forelesninger/Uke 9/Kode/test_gruppe.py"

Navn: Gro H. Brundtland. Alder: 50
Navn: Jonas G. Støre. Alder: 40
```


hovedprogram
gruppe_1
Gruppe instance
_gruppe_nr
1
_medlemmer
list
0
1
Person instance
Navn: Gro H. Brundtland. Alder: 50
_alder
50
_navn
Person instance
Navn: Jonas G. Støre. Alder: 40
_alder
40
_navn
Navn instance
Gro H. Brundtland
_etter
"Brundtland"
_fornavn
"Gro"
_mellom
"H."
Navn instance
Jonas G. Støre
_etter
"Støre"
_fornavn
"Jonas"
_mellom
"G."

OBS: Hvilken *type* har argumenter og returverdier?

- Vær oppmerksom på
  - hvilke typer som forventes som argumenter til funksjoner og metoder
  - hvilken type som returneres (om noen)

- Eks:
  - Returnerer metoden hent_navn ...
    .. En string? Et Navn-objekt? En liste med alle for-/ mellom- og etternavn? Flere navn av en slik type?

- Feil som lett skaper problemer!
  - sende inn feil type argument (for eksempel en string som navn ved opprettelse av ny Person)
  - bruke en returverdi fra f eks hent_navn-metoden i Person som en string i stedet for som et Navn-objekt

- Hjelp til selvhjelp
  - Bruk variabelnavn som minner deg (og andre) på hva de inneholder, inkludert tips om type!
  - string eller referanse, ett eller flere elementer, ..
  - Eksempler: person, personer
  - Typeannotasjoner (*type hints*)

```
class Person:
  def __init__(self, fullt_navn: Navn, alder: int):
```

Verdien None

- Verdien None signaliserer at en referansevariabel ikke holder på noe objekt i øyeblikket.
- None *kan* være en nyttig initialverdi om en (instans)variabel skal få verdi senere eller en returverdi om en metode ikke finner et objekt
- Er da nødvendig å sjekke mot None for å unngå å kalle på en metode i et objekt som ikke finnes (får ellers feil under kjøring)
- Testes med is eller is not (ikke == eller !=)

```
rek3 = None   # Kun for illustrasjon
:
:
areal = rek3.areal()
```

kjøretidsfeil!

```
if rek3 is not None:
  areal = rek3.areal()
```

ok, blir ikke utført

Oppsummert spesielle (magiske) metoder

- Er ikke innebygget i Python fordi de avhenger av semantikken i den enkelte klassen – som bare du som programmerer klassen kan bestemme
- Ofte nyttig å skrive `__str__` (og noen ganger `__eq__`) for egne klasser
- Kan være behov for spesielle metoder for andre logiske operatorer (<, >, !=, etc) – for eksempel for å sortere (__lt__, __gt__, __ne__)
- Skal aldri kalles direkte/ med navn. For eksempel
  - ikke ~~rek.__str__()~~, men str(rek) eller print(rek)
  - ikke ~~rek1.__eq__(rek2)~~, men rek1 == rek2

Neste uke

- Nye anvendelser av det vi har vært gjennom
- Todimensjonale strukturer – representasjon og visualisering
- Viktige strukturer i informatikk: Lenkelister, trær, grafer
- Livekoding av kollektiv-rutenettet i Oslo