

# Objektorientert programmering i Python: Introduksjon

IN1000

Høst 2021 – uke 7

Siri Moe Jensen

# Læringsmål uke 7

- Kjenne til motivasjon og bakgrunn for objektorientert programmering
- Kunne definere en klasse, opprette og bruke objekter
- Forstå sentrale begreper som grensesnitt og innkapsling
- Kjenne til utviklingsprosessen for en klasse gjennom design, implementasjon og testing

# Forrige uke

- Repetisjon, fordypning, noen andre perspektiver
- Hva er et "godt program"?
  - kjører og gjør det det skal, evt feiler på en "god" måte
  - men også lesbart, velstrukturert -> lett å sette seg inn i og utvide/ endre
- Mange valg i programmering handler om annet enn absolutter som syntaks og semantikk! I IN1000 prøver vi å formidle noen, relativt enkle, generelle prinsipper og kriterier

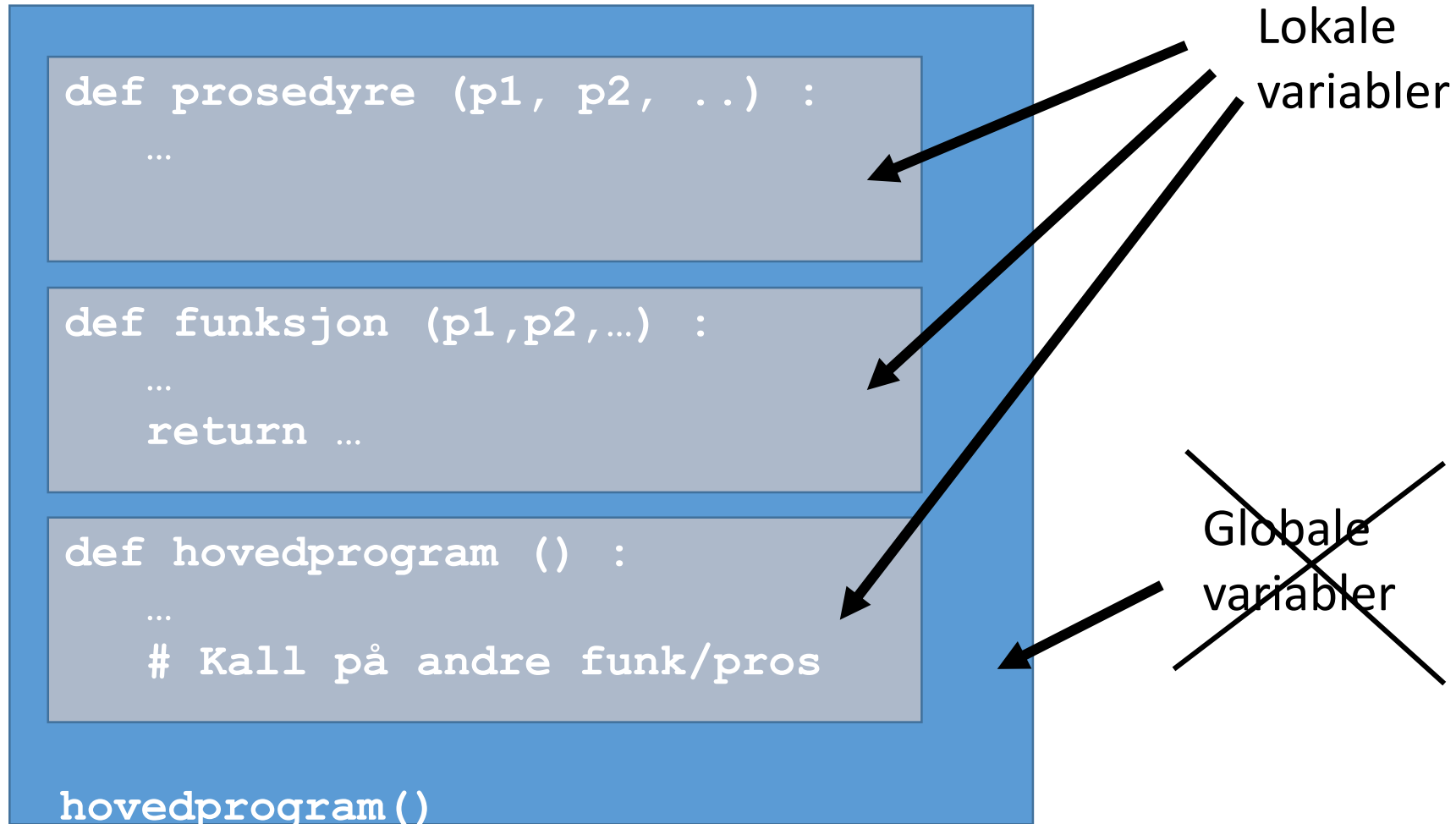
###

- Dessuten: Bli kjent med et tankesett/ paradigme som støtter systematisk tilnærming og strukturering av både problem og kode:

**Objektorientert programmering.**

# Lite tilbakeblikk: Programflyt og skop

mittProgram.py



# Lite tilbakeblikk: Funksjoner er uttrykk

- Prosedyre tar 0-n argumenter – returnerer ingen verdi
- Funksjoner tar 0-n argumenter – returnerer én verdi
  
- Når vi trenger en verdi i et program har vi flere muligheter:
  - er det alltid den samme? Literal/ konstant
  - Har vi lagret verdien i en variabel?
  - Kan vi regne den ut?
  - Kan vi finne den ved å **kalle på en funksjon?**

*Uttrykk*

# Lite tilbakeblikk: Funksjoner er uttrykk

- Funksjoner kan kombineres:

```
alder = int(input("Din alder: "))
```



- funksjonen `input` returnerer en verdi som brukes direkte som argument til funksjonen `int`.
- `int` returnerer en verdi som legges i variabelen `alder`.

# Hvorfor objektorientert programmering?

# Bakgrunn

- Store og komplekse systemer etter ~20 år programmering med (noen av) de mekanismene dere har lært hittil.
- Hovedsakelig matematiske beregninger – komplekse, men formelt definerte - store tallmengder, men repetitive operasjoner.
- Kompleksitet ga høye kostnader og feil. Arbeidsdeling og gjenbruk ble vanskelig.

$$\begin{aligned}
 \mathcal{L}_{SM} = & -\frac{1}{2}\partial_\nu g_\mu^a \partial_\nu g_\mu^a - g_s f^{abc} \partial_\mu g_\nu^a g_\mu^b g_\nu^c - \frac{1}{4}g_s^2 f^{abc} f^{ade} g_\mu^b g_\nu^c g_\mu^d g_\nu^e - \partial_\nu W_\mu^+ \partial_\nu W_\mu^- - \\
 & M^2 W_\mu^+ W_\mu^- - \frac{1}{2}\partial_\nu Z_\mu^0 \partial_\nu Z_\mu^0 - \frac{1}{2c_w^2} M^2 Z_\mu^0 Z_\mu^0 - \frac{1}{2}\partial_\mu A_\nu \partial_\mu A_\nu - ig c_w (\partial_\nu Z_\mu^0 (W_\mu^+ W_\nu^- - W_\nu^+ W_\mu^-) - \\
 & Z_\nu^0 (W_\mu^+ \partial_\nu W_\mu^- - W_\mu^- \partial_\nu W_\mu^+) + Z_\mu^0 (W_\nu^+ \partial_\nu W_\mu^- - W_\nu^- \partial_\nu W_\mu^+)) - ig s_w (\partial_\nu A_\mu (W_\mu^+ W_\nu^- - \\
 & W_\nu^+ W_\mu^-) - A_\nu (W_\mu^+ \partial_\nu W_\mu^- - W_\mu^- \partial_\nu W_\mu^+) + A_\mu (W_\nu^+ \partial_\nu W_\mu^- - W_\nu^- \partial_\nu W_\mu^+)) - \\
 & \frac{1}{2}g^2 W_\mu^+ W_\mu^- W_\nu^+ W_\nu^- + \frac{1}{2}g^2 W_\mu^+ W_\nu^- W_\mu^- W_\nu^+ + g^2 c_w^2 (Z_\mu^0 W_\nu^+ Z_\nu^0 W_\mu^- - Z_\mu^0 Z_\nu^0 W_\nu^+ W_\mu^-) + \\
 & g^2 s_w^2 (A_\mu W_\nu^+ A_\nu W_\mu^- - A_\mu A_\nu W_\nu^+ W_\mu^-) + g^2 s_w c_w (A_\mu Z_\nu^0 (W_\mu^+ W_\nu^- - W_\nu^+ W_\mu^-) - \\
 & 2A_\mu Z_\mu^0 W_\nu^+ W_\nu^-) - \frac{1}{2}\partial_\mu H \partial_\mu H - 2M^2 \alpha_h H^2 - \partial_\mu \phi^+ \partial_\mu \phi^- - \frac{1}{2}\partial_\mu \phi^0 \partial_\mu \phi^0 - \\
 & \beta_h \left( \frac{2M^2}{g^2} + \frac{2M}{g} H + \frac{1}{2}(H^2 + \phi^0 \phi^0 + 2\phi^+ \phi^-) \right) + \frac{2M^4}{g^2} \alpha_h - g \alpha_h M (H^3 + H \phi^0 \phi^0 + 2H \phi^+ \phi^-) - \\
 & \frac{1}{8}g^2 \alpha_h (H^4 + (\phi^0)^4 + 4(\phi^+ \phi^-)^2 + 4(\phi^0)^2 \phi^+ \phi^- + 4H^2 \phi^+ \phi^- + 2(\phi^0)^2 H^2) - g M W_\mu^+ W_\mu^- H - \\
 & \frac{1}{2}g \frac{M}{c_w^2} Z_\mu^0 Z_\mu^0 H - \frac{1}{2}ig (W_\mu^+ (\phi^0 \partial_\mu \phi^- - \phi^- \partial_\mu \phi^0) - W_\mu^- (\phi^0 \partial_\mu \phi^+ - \phi^+ \partial_\mu \phi^0)) + \\
 & \frac{1}{2}g (W_\mu^+ (H \partial_\mu \phi^- - \phi^- \partial_\mu H) + W_\mu^- (H \partial_\mu \phi^+ - \phi^+ \partial_\mu H)) + \frac{1}{2}g \frac{1}{c_w} (Z_\mu^0 (H \partial_\mu \phi^0 - \phi^0 \partial_\mu H) + \\
 & M (\frac{1}{c_w} Z_\mu^0 \partial_\mu \phi^0 + W_\mu^+ \partial_\mu \phi^- + W_\mu^- \partial_\mu \phi^+)) - ig \frac{2M}{c_w} M Z_\mu^0 (W_\mu^+ \phi^- - W_\mu^- \phi^+) + ig s_w M A_\mu (W_\mu^+ \phi^- - \\
 & W_\mu^- \phi^+) - ig \frac{1-2c_w^2}{2c_w} Z_\mu^0 (\phi^+ \partial_\mu \phi^- - \phi^- \partial_\mu \phi^+) + ig s_w A_\mu (\phi^+ \partial_\mu \phi^- - \phi^- \partial_\mu \phi^+) - \\
 & \frac{1}{4}g^2 W_\mu^+ W_\mu^- (H^2 + (\phi^0)^2 + 2\phi^+ \phi^-) - \frac{1}{8}g^2 \frac{1}{c_w} Z_\mu^0 Z_\mu^0 (H^2 + (\phi^0)^2 + 2(2s_w^2 - 1)^2 \phi^+ \phi^-) - \\
 & \frac{1}{2}g^2 \frac{2c_w^2}{c_w} Z_\mu^0 \phi^0 (W_\mu^+ \phi^- + W_\mu^- \phi^+) - \frac{1}{2}ig^2 \frac{2c_w^2}{c_w} Z_\mu^0 H (W_\mu^+ \phi^- - W_\mu^- \phi^+) + \frac{1}{2}g^2 s_w A_\mu \phi^0 (W_\mu^+ \phi^- + \\
 & W_\mu^- \phi^+) + \frac{1}{2}ig^2 s_w A_\mu H (W_\mu^+ \phi^- - W_\mu^- \phi^+) - g^2 \frac{2c_w^2}{c_w} (2c_w^2 - 1) Z_\mu^0 A_\mu \phi^+ \phi^- - g^2 s_w^2 A_\mu A_\mu \phi^+ \phi^- + \\
 & \frac{1}{2}ig_s \lambda_{ij}^a (\tilde{q}_i^\sigma \gamma^\mu q_j^\sigma) g_\mu^a - \tilde{e}^\lambda (\gamma^\mu + m_e^\lambda) e^\lambda - \tilde{\nu}^\lambda (\gamma^\mu + m_\nu^\lambda) \nu^\lambda - \tilde{u}_j^\lambda (\gamma^\mu + m_u^\lambda) u_j^\lambda - \tilde{d}_j^\lambda (\gamma^\mu + m_d^\lambda) d_j^\lambda + \\
 & ig s_w A_\mu \left( -(\tilde{e}^\lambda \gamma^\mu e^\lambda) + \frac{2}{3}(\tilde{u}_j^\lambda \gamma^\mu u_j^\lambda) - \frac{1}{3}(\tilde{d}_j^\lambda \gamma^\mu d_j^\lambda) \right) + \frac{ig}{4c_w} Z_\mu^0 \{ (\tilde{\nu}^\lambda \gamma^\mu (1 + \gamma^5) \nu^\lambda) + (\tilde{e}^\lambda \gamma^\mu (4s_w^2 - \\
 & 1 - \gamma^5) e^\lambda) + (\tilde{d}_j^\lambda \gamma^\mu (\frac{4}{3}s_w^2 - 1 - \gamma^5) d_j^\lambda) + (\tilde{u}_j^\lambda \gamma^\mu (1 - \frac{8}{3}s_w^2 + \gamma^5) u_j^\lambda) \} + \\
 & \frac{ig}{2\sqrt{2}} W_\mu^+ \left( (\tilde{\nu}^\lambda \gamma^\mu (1 + \gamma^5) U^{lep}{}_{\lambda\kappa} e^\kappa) + (\tilde{u}_j^\lambda \gamma^\mu (1 + \gamma^5) C_{\lambda\kappa} d_j^\kappa) \right) + \\
 & \frac{ig}{2\sqrt{2}} W_\mu^- \left( (\tilde{e}^\kappa U^{lep}{}_{\kappa\lambda} \gamma^\mu (1 + \gamma^5) \nu^\lambda) + (\tilde{d}_j^\kappa C_{\kappa\lambda}^\dagger \gamma^\mu (1 + \gamma^5) u_j^\lambda) \right) + \\
 & \frac{ig}{2M\sqrt{2}} \phi^+ (-m_e^\kappa (\tilde{\nu}^\lambda U^{lep}{}_{\lambda\kappa} (1 - \gamma^5) e^\kappa) + m_\nu^\lambda (\tilde{\nu}^\lambda U^{lep}{}_{\lambda\kappa} (1 + \gamma^5) e^\kappa) + \\
 & \frac{ig}{2M\sqrt{2}} \phi^- (m_e^\lambda (\tilde{e}^\lambda U^{lep}{}_{\lambda\kappa}^\dagger (1 + \gamma^5) \nu^\kappa) - m_\nu^\kappa (\tilde{e}^\lambda U^{lep}{}_{\lambda\kappa}^\dagger (1 - \gamma^5) \nu^\kappa) - \frac{2}{M} m_\nu^2 H (\tilde{\nu}^\lambda \nu^\lambda) - \\
 & \frac{g}{2} \frac{m_\nu^\lambda}{M} H (\tilde{e}^\lambda e^\lambda) + \frac{ig}{2} \frac{m_\nu^\lambda}{M} \phi^0 (\tilde{\nu}^\lambda \gamma^5 \nu^\lambda) - \frac{ig}{2} \frac{m_\nu^\lambda}{M} \phi^0 (\tilde{e}^\lambda \gamma^5 e^\lambda) - \frac{1}{4} \tilde{\nu}_\lambda M_{\lambda\kappa}^R (1 - \gamma_5) \tilde{\nu}_\kappa - \\
 & \frac{1}{4} \tilde{\nu}_\lambda M_{\lambda\kappa}^R (1 - \gamma_5) \tilde{\nu}_\kappa + \frac{ig}{2M\sqrt{2}} \phi^+ (-m_d^\kappa (\tilde{u}_j^\lambda C_{\lambda\kappa} (1 - \gamma^5) d_j^\kappa) + m_u^\lambda (\tilde{u}_j^\lambda C_{\lambda\kappa} (1 + \gamma^5) d_j^\kappa) + \\
 & \frac{ig}{2M\sqrt{2}} \phi^- (m_d^\lambda (\tilde{d}_j^\lambda C_{\lambda\kappa}^\dagger (1 + \gamma^5) u_j^\kappa) - m_u^\kappa (\tilde{d}_j^\lambda C_{\lambda\kappa}^\dagger (1 - \gamma^5) u_j^\kappa) - \frac{g}{2} \frac{m_u^\lambda}{M} H (\tilde{u}_j^\lambda u_j^\lambda) - \frac{g}{2} \frac{m_d^\lambda}{M} H (\tilde{d}_j^\lambda d_j^\lambda) + \\
 & \frac{ig}{2} \frac{m_u^\lambda}{M} \phi^0 (\tilde{u}_j^\lambda \gamma^5 u_j^\lambda) - \frac{ig}{2} \frac{m_d^\lambda}{M} \phi^0 (\tilde{d}_j^\lambda \gamma^5 d_j^\lambda)
 \end{aligned}$$



# Hva skjedde ~1967?

Samtidig:

- Datamaskinene ble kraftigere
- billigere
- mer utbredt

=> Nye anvendelser!

- Helsevesenet
- Trafikk-planlegging
- Økonomisystemer
  
- Facebook??



# En måte å strukturere programmer på

- *Prosedyrer og funksjoner* kan brukes for å løse deloppgaver
- Hva om oppgavene handler om å bearbeide felles, kanskje komplekse, data?
- For eksempel tidspunkter:
  - Dato: dag, måned, år
  - Klokkeslett: Time, minutt, sekund, ....
  - Ulike formater: a.m. og p.m. eller 24-timers
  - Ulike måter å skrive på: MM/DD/YY eller DD.MM.ÅÅ eller ...
  - Endre fra ett format til et annet
  - Sjekke hvilket tidspunkt som kommer først
  - Regne ut hvor lang tid det er mellom to tidspunkter
  - hva heter måned nummer 3
- Mange variabler, mange funksjoner, lett å miste oversikt

# En måte å strukturere programmer på

- *Prosedyrer og funksjoner* kan brukes for å løse deloppgaver
- 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*

Gjenstår: Hvilke data og operasjoner trengs? Hvordan gruppere i objekter?

# Et objekt representerer "noe"

Fysisk, konkret eller abstrakt

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

# Konseptet objektorientering

- 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

# Å programmere med objekter

Objekter er *instanser* av en klasse

# Vi har tidligere brukt objekter som

- ..representerer lister m/ tilhørende metoder (tjenester)

```
navneliste = []  
navneliste.append("Per")  
navneliste.append("Paal")  
navneliste.append("Espen")  
navn = navneliste.pop()
```

- ..representerer filer m/ tilhørende metoder

```
utfil = open ("navn.txt", "w")  
utfil.write (navn)  
utfil.close()
```

# 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"

```
minListe.append(nyttElem)
```

- Vi utfører metoder på objektet vårt ved hjelp av kallet *< objekt.metode(parametere) >*
- En metode kan ta argumenter
- En metode kan returnere en verdi (som en funksjon)



# Grensesnitt

- Et objekt tilbyr *ett sett tjenester* i form av metoder som programmerere kan kalle på for å lese av, endre eller aktivisere et objekt: *Objektets grensesnitt*
- Grensesnittet bestemmes av *objektets klasse*
- Hvordan finner vi ut hvilke metoder som tilbys av objekter av en bestemt klasse?
- F eks i læreboka eller Python dokumentasjon
  - google search: python methods list
  - f eks <https://docs.python.org/3/tutorial/datastructures.html>  
(nb bl.a. ulike versjoner av Python)
  - python.org (presist og korrekt) eller diverse kurs/ nettsteder (kan være mer lettlest)

# Uformell beskrivelse av grensesnitt

## **for et liste-objekt (f. eks. handleliste)**

- legg til et element
- les av første element
- fjern et element
- ...

## **for et dato-objekt (f.eks. dagens dato)**

- hva er navnet på måneden i datoen?
- gi meg en lesbar streng av datoen
- endre datoen til neste dag
- er dato xx.xx.xx før eller etter dette objektet?
- ...

# Et objekt har..

- et **grensesnitt**



- og en innmat:  
**Implementasjon**



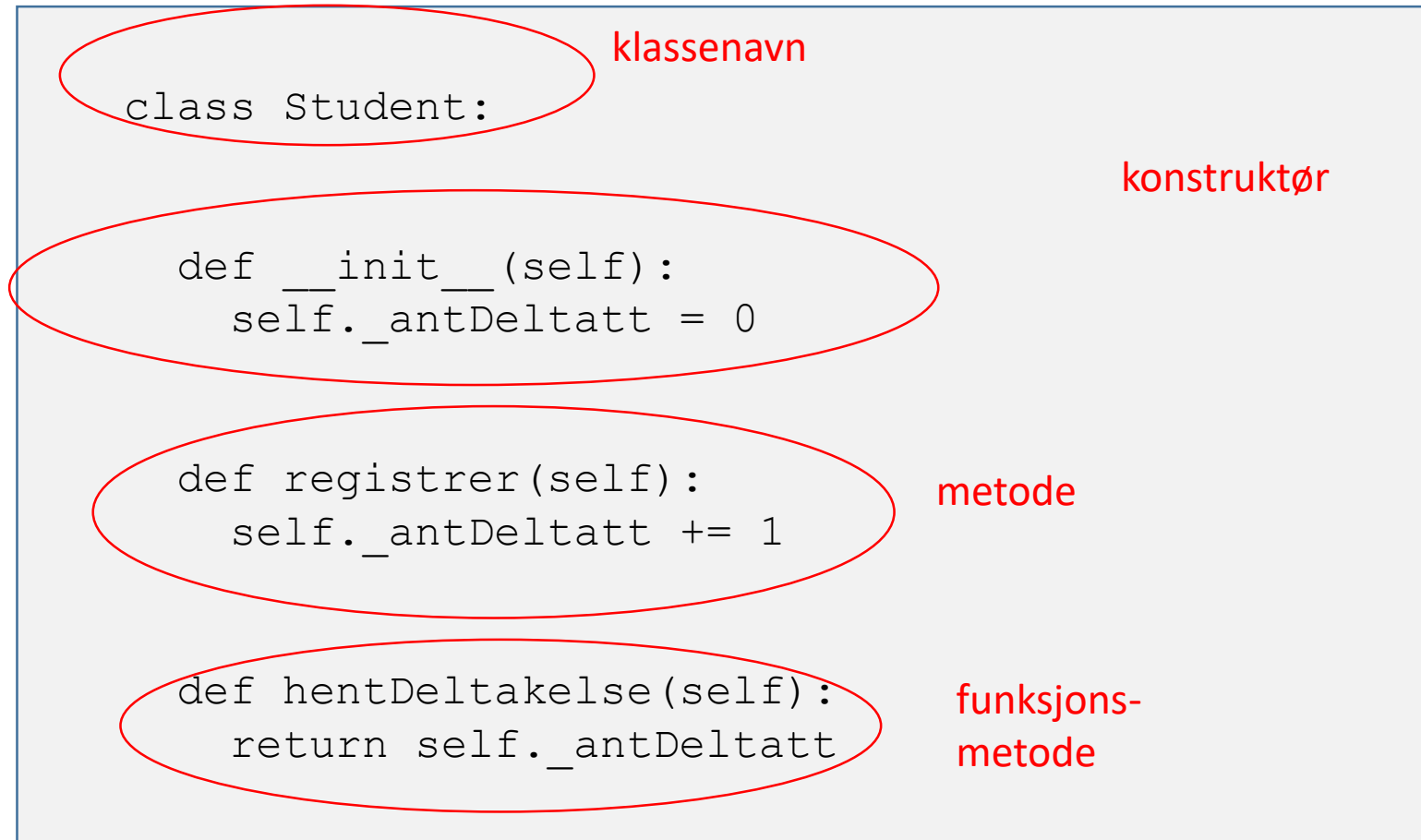
# Å lage egne klasser

Hva finnes bak grensesnittet?

# Klasser

- En klasse er et mønster/ oppskrift for objekter av samme type. Eks:
  - Student
  - Dato
- Klassedefinisjonen definerer hvilke metoder objektene skal tilby (grensesnittet) og hvordan disse implementeres:
  - Hvilke data hvert objekt inneholder (i *instansvariabler*)
  - Innholdet i metodene (som opererer på instansvariablene i objektet)
- Du kan definere (programmere) dine egne klasser som du så kan opprette en eller flere instanser (objekter) av

# Klasse-definisjon (syntaks)



# Opprette objekter (syntaks)

- Noen innebygde klasser har snarveier for opprettelse av objekter
  - `""` oppretter (og returnerer) en ny **str**
  - `[]` oppretter (og returnerer) en ny **list**
- Ellers opprettes (og returneres) objekter vha klassenavn med argumentliste (som kan være tom):

```
nyStudent = Student()
```

# En ny klasse: Fremgangsmåte

Eksempel Student



# Eksempel Student

- 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?
  - => Hjelp for gruppelærer:  
Registrere hver student's oppmøte i gruppetimene (ved krav til min. oppmøte)

# Student: Starter "utenfra"

## Uformelt grensesnitt

- Ønsker å kunne gjøre følgende:
  - Lage Student-objekt med deltakelse 0 ved semesterstart
  - Øke oppmøte med 1 når studenten deltar på time
  - Lese av (hente ut) antall ganger deltatt for en student
- Dvs: Grensesnittet må (i tillegg til konstruktør) tilby metoder som:
  - Registrerer deltakelse
  - Henter antall ganger deltatt

# Student: Grensesnitt i Python

```
## Behandler en students deltakelse på gruppetimer
#
class Student:
    ## Oppretter student med deltakelse 0
    #
    def __init__(self):

        ## Registrer deltakelse
        #
        def registrer(self):

            ## Hent ut deltakelse
            #
            def hentDeltakelse(self):
```

# Student: Implementasjon av klassen

```
class Student:

    def __init__(self):
        self._antDeltatt = 0

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

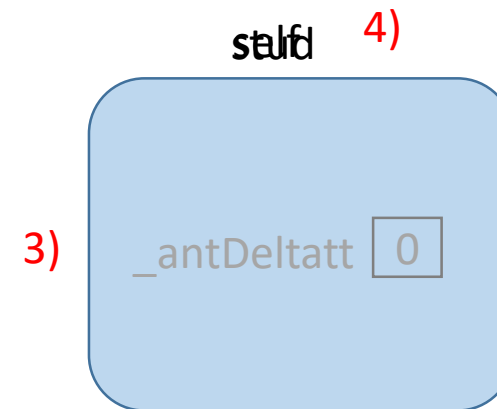
    def hentDeltakelse(self):
        return self._antDeltatt
```

# Student: Opprette et objekt av klassen

- **Instansvariabler** opprettes og initialiseres i `__init__` metoden (konstruktøren)
- Konstruktøren kalles automatisk når et nytt objekt opprettes:
  1. oppretter objektet
  2. oppretter og initialiserer instansvariablene i objektet
  3. returnerer det nye objektet til kallstedet

```
class Student:  
    2) def __init__(self):  
        3) self._antDeltatt = 0  
  
stud = Student() 1)
```

4)



# Student: Opprette et objekt av klassen


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

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

    def hentDeltakelse(self):
        return self._antDeltatt

stud = Student()
stud.registrer()
antall = stud.hentDeltakelse()
```

NB: Vi oppgir ikke noe argument til parameteren 'self'!



# Oppgave

```
class Student:

    def __init__(self):
        self._antDeltatt = 0

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

    def hentDeltakelse(self):
        return self._antDeltatt
```

1. Utvid klassen student med studentens navn:
  - endre konstruktør: Ny parameter `navn`
  - instansvariabel `_navn`
  - ny funksjonsmetode `hentNavn`
2. Hvordan opprettes nå et nytt Student-objekt?

# Student-klasse utvidet med navn: Implementasjon i Python

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

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

    def hentDeltakelse(self):
        return self._antDeltatt

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



# Test av klassen

- Vi skriver et testprogram i en egen fil
- Importerer klassen Student (fra filen student.py)
- 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

# Testprogram for Student-klassen

```
from student import Student

stud1 = Student("Siri")
stud2 = Student("Geir Kjetil")
stud3 = Student("Henrik")

stud1.registrer()
stud1.registrer()
stud2.registrer()

# Sjekk at registrert deltakelse er som forventet
assert(stud1.hentDeltakelse() == 2)
assert(stud2.hentDeltakelse() == 1)
assert(stud3.hentDeltakelse() == 0)
```

## testStudent.py (alternativ test)

```
from student import Student

stud1 = Student("Siri")
stud2 = Student("Geir Kjetil")
stud3 = Student("Henrik")

stud1.registrer()
stud1.registrer()
stud2.registrer()

# Alternativ test. Lettere å debugge, men må fjernes:

print (stud1.hentNavn() + ": " + str(stud1.hentDeltakelse()))
print (stud2.hentNavn() + ": " + str(stud2.hentDeltakelse()))
print (stud3.hentNavn() + ": " + str(stud3.hentDeltakelse()))
```

# Kjøring alternativt testprogram

```
M:\Ifi\Undervisning\IN1000 H2020\Forelesninger\Uke7\Prog
>python testStudent.py
Siri: 2
Geir Kjetil: 1
Henrik: 0

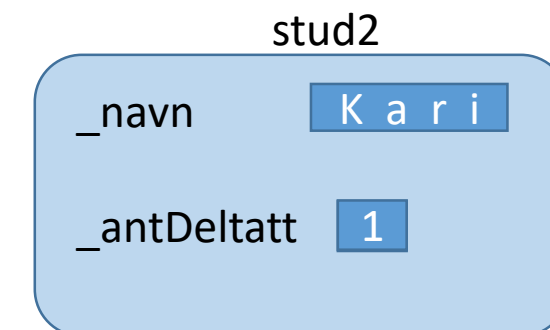
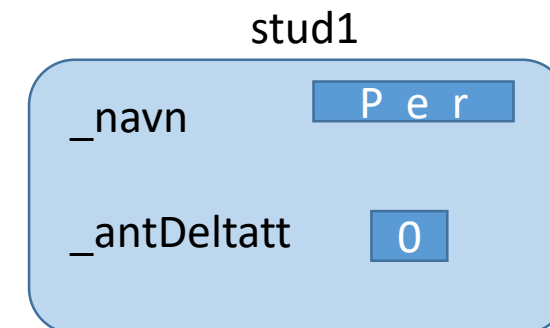
M:\Ifi\Undervisning\IN1000 H2020\Forelesninger\Uke7\Prog
>
```

<programfiler ligger under uke7 på semestersiden>

# Studentobjekter

- Vi har tidligere sett objektene vi bruker "fra utsiden", og nøydt oss med å bruke tjenestene deres
- Når vi lager egne klasser, bestemmer vi hvordan "innmaten" i objektene skal se ut.
- Kan være nyttig å tegne, spesielt når vi får mange objekter av forskjellige klasser



```
stud1 = Student("Per")  
stud2 = Student("Kari")  
stud2.registrer()
```



# Hvorfor lage en klasse?

- 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 flere like objekter med samme data og oppførsel (eks representere studenter)
- For å simulere sammensatte fenomener vha objekter som representerer virkelige elementer (eks trafikk- simuleringer)

# Fremgangsmåte: OOP

1. Identifiser aktuelle klasser (hva er sentrale «ting»/ begreper programmet skal behandle?)
2. Design klassens grensesnitt (hvilke tjenester ønsker jeg fra objekter av klassen?)
3. Design klassens datarepresentasjon (hvordan skal objektene representere dataene sine?)
4. Implementer (fyll ut) metodene (skriv klassen ferdig)  

5. Lag et testprogram som oppretter et eller flere objekter og kaller på metodene i objektene's grensesnitt  


# Innkapsling

- Prinsipp som innebærer at programmerere kun har tilgang til (utvalgte) metoder i objekter av en klasse: Klassens grensesnitt ("public interface")
- Data og evt andre metoder (hjelpemetoder) definert inne i klassen er "non-public" og brukes kun inne i klassedefinisjonen
- Sentralt prinsipp i objektorientert programmering
  - en klasse kan endres så lenge grensesnittet er det samme – uten at det får konsekvenser der klassen brukes
  - gjennomført ansvarsdeling (splitt og hersk)



# Innkapsling i Python og IN1000

- Python støtter innkapsling med *konvensjoner*, men har ikke mekanismer for å hindre tilgang
  - alle instansvariable og metoder som ikke skal inngå i grensesnittet, får navn som begynner med `_`
- Dette prinsippet skal dere bruke konsekvent i IN1000
  
- I praktisk bruk er det et *designvalg* om / når/ hvordan man anvender innkapsling
- Dere vil se eksempler på direkte aksess av data (utenom klassens metoder) blant annet i IN1010

# Eksempel: Navn

- 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 (Støre, Jonas Gahr)

# Klassen Navn

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
```

# Testprogram for klassen Navn

Filen `testnavn.py`

```
from navn import Navn

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

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

    navn3 = Navn("Siri", "Moe", "Jensen")

    assert (navn1.sortert() != navn2.sortert())
    assert (navn1.sortert() == navn3.sortert())

hovedprogram()
```

*Hva skjer her?*

*Og her?*

navn1

```
_fornavn = Siri
_mellom = Moe
_etter = Jensen
```

navn2

```
_fornavn = Geir
_mellom = Kjetil
_etter = Sandve
```

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

# OO støtter 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 på metoder

```
class Student:
    def __init__(self):
        self._antDeltatt = 0
    def registrer(self):
        self._antDeltatt = self.antDeltatt + 1
    def hentDeltakelse(self):
        return self._antDeltatt
stud1 = Student()
stud1.registrer()
```

klassenavn

konstruktør

metode

funksjons-metode

metodekall på objektet stud1

# Åpne menti.com, kode 8759 1006

- Bruk resten av tiden i dag til å svare 😊
- Men den er åpen til torsdag morgen
- Du kan svare flere ganger og gå frem og tilbake i oppgavene
  
- Vi går gjennom oppgavene på forelesning neste uke