Vi starter 10:15. Imens, ta gjerne en titt på lenken fra uke7-siden under ressurser:
Introduksjon til objektorientert programmering (8 min video)

Objektorientert programmering i Python: Introduksjon

IN1000

Høst 2020 – 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 "ting å tenke på"

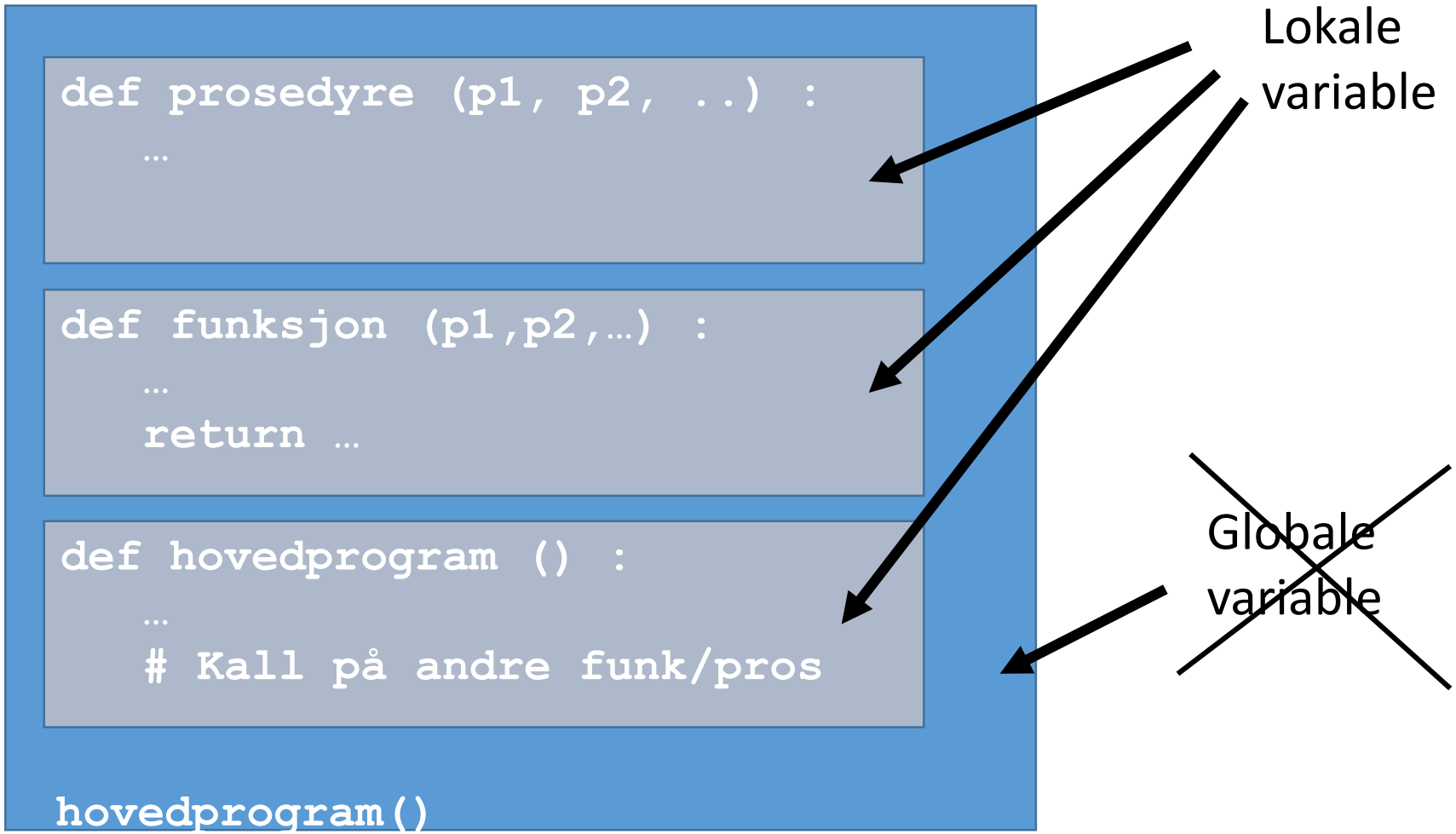
###

- 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

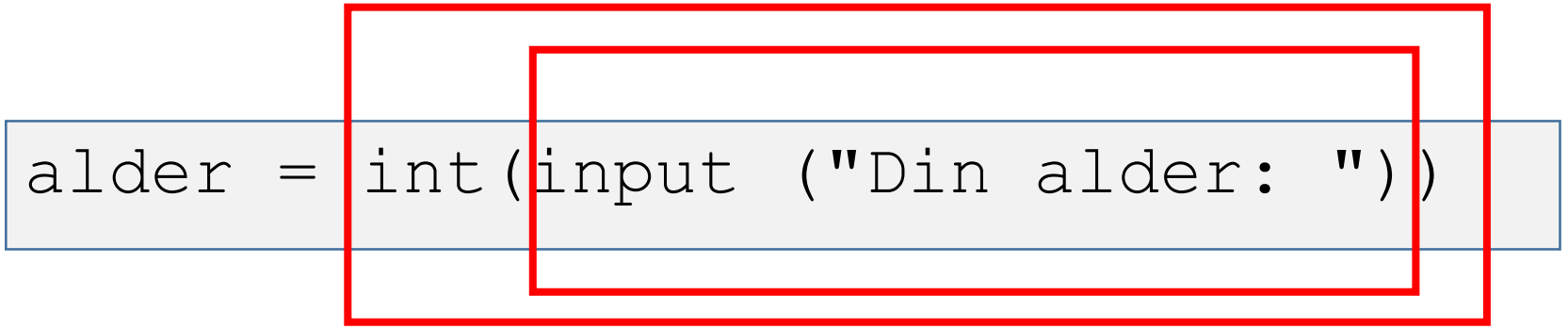
- Prosedyre tar 0-n argumenter – returnerer ingen verdi
- Funksjoner tar 0-n argumenter – returnerer én verdi

- 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:



- 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.

$$\mathcal{L}_{SM} = -\frac{1}{2}\partial_\nu g^a_\mu \partial_\nu g^a_\mu - g_s f^{abc}\partial_\mu g^a_\nu g^b_\mu g^c_\nu - \frac{1}{4}g_s^2 f^{abc}f^{ade} g^b_\mu g^c_\nu g^d_\mu g^e_\nu - \partial_\nu W^+_\mu \partial_\nu W^-_\mu - M^2 W^+_\mu W^-_\mu - \frac{1}{2}\partial_\nu Z^0_\mu \partial_\nu Z^0_\mu - \frac{1}{2c_w^2}M^2 Z^0_\mu Z^0_\mu - \frac{1}{2}\partial_\mu A_\nu \partial_\mu A_\nu - igc_w(\partial_\nu Z^0_\mu (W^+_\mu W^-_\nu - W^+_\nu W^-_\mu) - Z^0_\nu (W^+_\mu \partial_\nu W^-_\mu - W^-_\mu \partial_\nu W^+_\mu) + Z^0_\mu (W^+_\nu \partial_\nu W^-_\mu - W^-_\nu \partial_\nu W^+_\mu)) - igs_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^0_\mu W^+_\mu Z^0_\nu W^-_\nu - Z^0_\mu Z^0_\mu W^+_\nu W^-_\nu) + g^2 s_w^2 (A_\mu W^+_\mu A_\nu W^-_\nu - A_\mu A_\mu W^+_\nu W^-_\nu) + g^2 s_w c_w (A_\mu Z^0_\nu (W^+_\mu W^-_\nu - W^+_\nu W^-_\mu) - 2A_\mu Z^0_\mu 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\left(H^3 + H\phi^0\phi^0 + 2H\phi^+\phi^-\right) - \frac{1}{8}g^2\alpha_h\left(H^4 + (\phi^0)^4 + 4(\phi^+\phi^-)^2 + 4(\phi^0)^2\phi^+\phi^- + 4H^2\phi^+\phi^- + 2(\phi^0)^2H^2\right) - gMW^+_\mu W^-_\mu H - \frac{1}{2}g\frac{M}{c_w^2}Z^0_\mu Z^0_\mu H - \frac{1}{2}ig\left(W^+_\mu(\phi^0\partial_\mu\phi^- - \phi^-\partial_\mu\phi^0) - W^-_\mu(\phi^0\partial_\mu\phi^+ - \phi^+\partial_\mu\phi^0)\right) + \frac{1}{2}g\left(W^+_\mu(H\partial_\mu\phi^- - \phi^-\partial_\mu H) + W^-_\mu(H\partial_\mu\phi^+ - \phi^+\partial_\mu H)\right) + \frac{1}{2}g\frac{1}{c_w}(Z^0_\mu(H\partial_\mu\phi^0 - \phi^0\partial_\mu H) + M\left(\frac{1}{c_w}Z^0_\mu\partial_\mu\phi^0 + W^+_\mu\partial_\mu\phi^- + W^-_\mu\partial_\mu\phi^+\right) - ig\frac{s_w^2}{c_w}MZ^0_\mu(W^+_\mu\phi^- - W^-_\mu\phi^+) + igs_wMA_\mu(W^+_\mu\phi^- - W^-_\mu\phi^+) - ig\frac{1-2c_w^2}{2c_w}Z^0_\mu(\phi^+\partial_\mu\phi^- - \phi^-\partial_\mu\phi^+) + igs_wA_\mu(\phi^+\partial_\mu\phi^- - \phi^-\partial_\mu\phi^+) - \frac{1}{4}g^2W^+_\mu W^-_\mu\left(H^2 + (\phi^0)^2 + 2\phi^+\phi^-\right) - \frac{1}{8}g^2\frac{1}{c_w^2}Z^0_\mu Z^0_\mu\left(H^2 + (\phi^0)^2 + 2(2s_w^2 - 1)^2\phi^+\phi^-\right) - \frac{1}{2}g^2\frac{s_w^2}{c_w}Z^0_\mu\phi^0(W^+_\mu\phi^- + W^-_\mu\phi^+) - \frac{1}{2}ig^2\frac{s_w^2}{c_w}Z^0_\mu H(W^+_\mu\phi^- - W^-_\mu\phi^+) + \frac{1}{2}g^2s_wA_\mu\phi^0(W^+_\mu\phi^- + W^-_\mu\phi^+) + \frac{1}{2}ig^2s_wA_\mu H(W^+_\mu\phi^- - W^-_\mu\phi^+) - g^2\frac{s_w}{c_w}(2c_w^2 - 1)Z^0_\mu A_\mu\phi^+\phi^- - g^2s_w^2A_\mu A_\mu\phi^+\phi^- + \frac{1}{2}ig_s\lambda^a_{ij}(\bar{q}^\sigma_i\gamma^\mu q^\sigma_j)g^a_\mu - \bar{e}^\lambda(\gamma\partial + m^\lambda_e)e^\lambda - \bar{\nu}^\lambda(\gamma\partial + m^\lambda_\nu)\nu^\lambda - \bar{u}^\lambda_j(\gamma\partial + m^\lambda_u)u^\lambda_j - \bar{d}^\lambda_j(\gamma\partial + m^\lambda_d)d^\lambda_j + igs_wA_\mu\left(-(\bar{e}^\lambda\gamma^\mu e^\lambda) + \frac{2}{3}(\bar{u}^\lambda_j\gamma^\mu u^\lambda_j) - \frac{1}{3}(\bar{d}^\lambda_j\gamma^\mu d^\lambda_j)\right) + \frac{ig}{4c_w}Z^0_\mu\{(\bar{\nu}^\lambda\gamma^\mu(1+\gamma^5)\nu^\lambda) + (\bar{e}^\lambda\gamma^\mu(4s_w^2 - 1 - \gamma^5)e^\lambda) + (\bar{d}^\lambda_j\gamma^\mu(\frac{4}{3}s_w^2 - 1 - \gamma^5)d^\lambda_j) + (\bar{u}^\lambda_j\gamma^\mu(1 - \frac{8}{3}s_w^2 + \gamma^5)u^\lambda_j)\} + \frac{ig}{2\sqrt{2}}W^+_\mu\left((\bar{\nu}^\lambda\gamma^\mu(1+\gamma^5)U^{lep}_{\lambda\kappa}e^\kappa) + (\bar{u}^\lambda_j\gamma^\mu(1+\gamma^5)C_{\lambda\kappa}d^\kappa_j)\right) + \frac{ig}{2\sqrt{2}}W^-_\mu\left((\bar{e}^\kappa U^{lep\dagger}_{\kappa\lambda}\gamma^\mu(1+\gamma^5)\nu^\lambda) + (\bar{d}^\kappa_j C^\dagger_{\kappa\lambda}\gamma^\mu(1+\gamma^5)u^\lambda_j)\right) + \frac{ig}{2M\sqrt{2}}\phi^+\left(-m^\kappa_e(\bar{\nu}^\lambda U^{lep}_{\lambda\kappa}(1-\gamma^5)e^\kappa) + m^\lambda_\nu(\bar{\nu}^\lambda U^{lep}_{\lambda\kappa}(1+\gamma^5)e^\kappa)\right) + \frac{ig}{2M\sqrt{2}}\phi^-\left(m^\lambda_e(\bar{e}^\lambda U^{lep\dagger}_{\lambda\kappa}(1+\gamma^5)\nu^\kappa) - m^\kappa_\nu(\bar{e}^\lambda U^{lep\dagger}_{\lambda\kappa}(1-\gamma^5)\nu^\kappa)\right) - \frac{g}{2}\frac{m^\lambda_\nu}{M}H(\bar{\nu}^\lambda\nu^\lambda) - \frac{g}{2}\frac{m^\lambda_e}{M}H(\bar{e}^\lambda e^\lambda) + \frac{ig}{2}\frac{m^\lambda_e}{M}\phi^0(\bar{\nu}^\lambda\gamma^5\nu^\lambda) - \frac{ig}{2}\frac{m^\lambda_\nu}{M}\phi^0(\bar{e}^\lambda\gamma^5 e^\lambda) - \frac{1}{4}\bar{\nu}_\lambda M^R_{\lambda\kappa}(1-\gamma_5)\bar{\nu}_\kappa - \frac{1}{4}\bar{\nu}_\lambda M^R_{\lambda\kappa}(1-\gamma_5)\bar{\nu}_\kappa + \frac{ig}{2M\sqrt{2}}\phi^+\left(-m^\kappa_d(\bar{u}^\lambda_j C_{\lambda\kappa}(1-\gamma^5)d^\kappa_j) + m^\lambda_u(\bar{u}^\lambda_j C_{\lambda\kappa}(1+\gamma^5)d^\kappa_j\right) + \frac{ig}{2M\sqrt{2}}\phi^-\left(m^\lambda_d(\bar{d}^\lambda_j C^\dagger_{\lambda\kappa}(1+\gamma^5)u^\kappa_j) - m^\kappa_u(\bar{d}^\lambda_j C^\dagger_{\lambda\kappa}(1-\gamma^5)u^\kappa_j\right) - \frac{g}{2}\frac{m^\lambda_u}{M}H(\bar{u}^\lambda_j u^\lambda_j) - \frac{g}{2}\frac{m^\lambda_d}{M}H(\bar{d}^\lambda_j d^\lambda_j) + \frac{ig}{2}\frac{m^\lambda_u}{M}\phi^0(\bar{u}^\lambda_j\gamma^5 u^\lambda_j) - \frac{ig}{2}\frac{m^\lambda_d}{M}\phi^0(\bar{d}^\lambda_j\gamma^5 d^\lambda_j)$$

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 (f.eks. sjokolade)
- …

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?

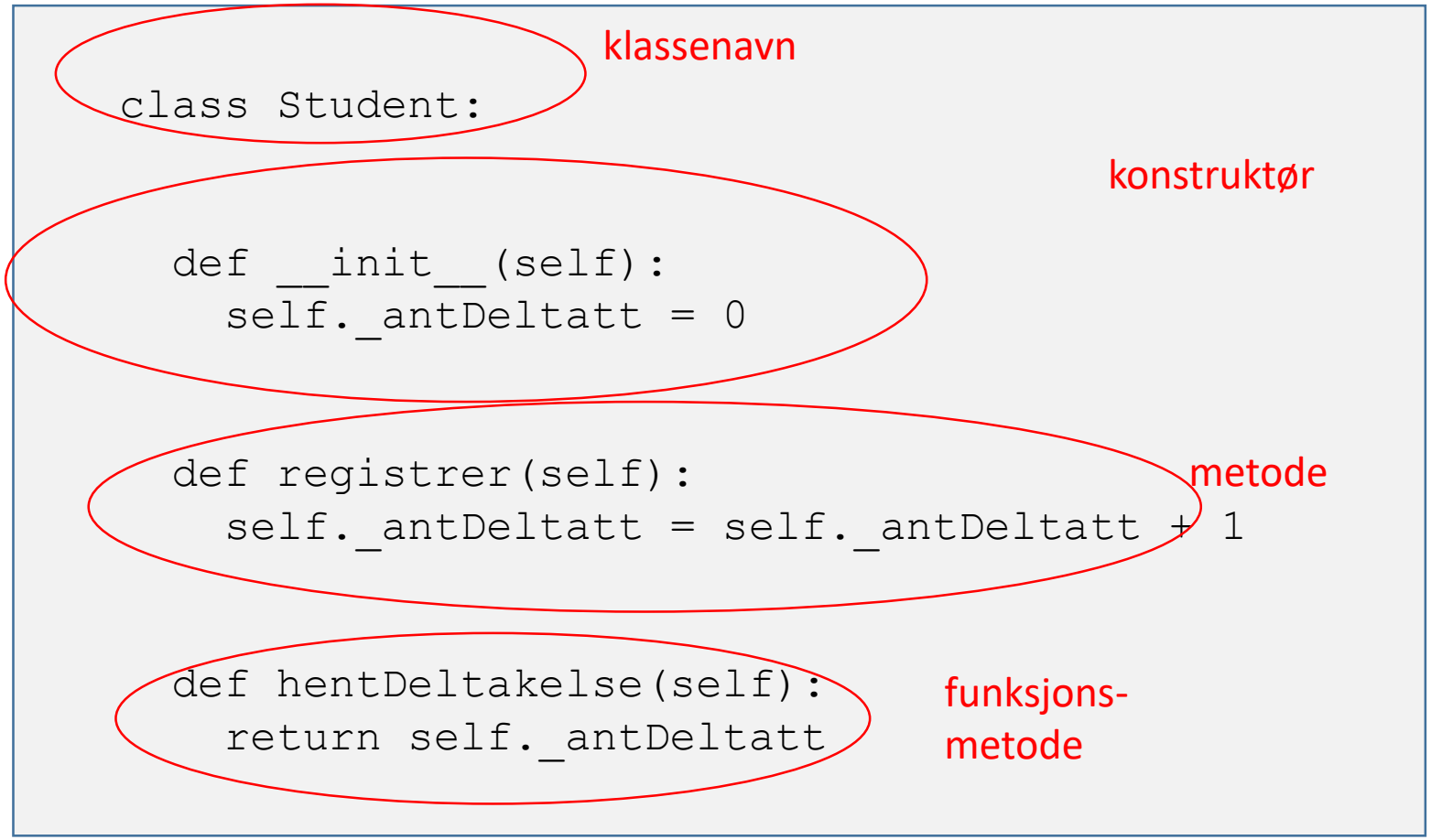
Klasser

- En klasse er et mønster/ oppskrift for objekter av samme type. Eks:
  - Student
  - Dato

- Klassedefinisjonen beskriver hvilke data hvert objekt skal ha (i *instansvariabler*) og de metodene som skal operere 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 en ny **`str`**
  - [] oppretter en ny **`list`**
- Ellers opprettes 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

- Hva skal jeg bruke klassen til?

  => Hjelp for gruppelærer:
  Holde rede på oppmøte i gruppetimene

Student: Grensesnitt uformelt

- Ønsker å kunne gjøre følgende:
  - Lage Student-objekt med null deltakelse ved semesterstart
  - Øke oppmøte med 1 ved deltakelse
  - Lese av (hente ut) deltakelse ved behov

- Grensesnittet må tilby metoder som:
  - Registrerer deltakelse
  - Henter antall ganger deltatt

Student: Grensesnitt i Python

```
## Behandler en students deltakelse på gruppetime
#
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 = self._antDeltatt + 1


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

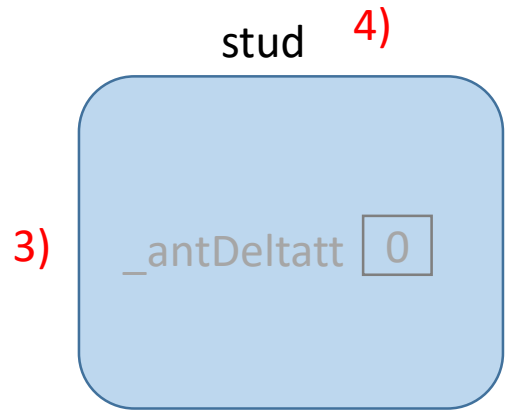
Student: Opprette et objekt av klassen

- **Instansvariabler** opprettes og initialiseres i __init__ metoden (konstruktøren)

- Konstruktøren kalles automatisk når et nytt objekt opprettes:
  1. oppretter objektet
  2. oppretter og initierer instansvariablene i objektet
  3. returnerer det nye objektet til kallstedet

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

stud = Student()
```

2) 3) 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

testStudent.py

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

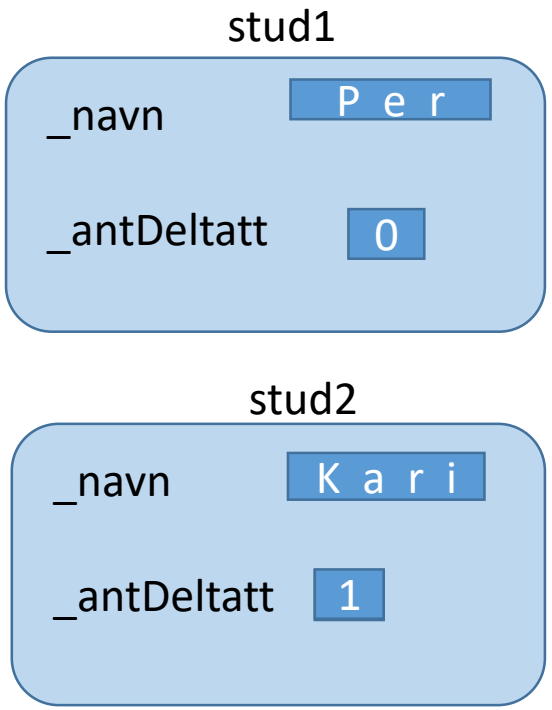
```
M:\Ifi\Undervisning\IN1000 H2020\Forelesninger\Uke7\Prog
>python testStudent.py
Siri: 2
Geir Kjetil: 1
Henrik: 0

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 nøyd 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 operasjoner trenger jeg for 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 objektenes 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 (Hareide, Knut Arild)
    - hente naturlig (Knut Arild Hareide)

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) skal få til objekter av klassen

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

- 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:
```

klassenavn

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

konstruktør

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

metode

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

funksjons-metode

```
stud1 = Student()
stud1.registrer()
```

metodekall på objektet stud1