Objektorientert programmering i Python: Introduksjon

IN1000

Høst 2023 – uke 7

Siri Moe Jensen

Læringsmål uke 7

Kjenne til motivasjon og bakgrunn for objektorientert programmering

Programmering: Kunne definere en klasse, opprette objekter og kalle metoder på objektene

Kunne bruke og forklare begrepene klasse, objekt, klassedefinisjon, konstruktør, metode og instansvariabel

Forstå sentrale begreper som grensesnitt og innkapsling

Kjenne til utviklingsprosessen for en klasse gjennom design, implementasjon og testing

Fra underveisevaluering – hva er vanskelig?

Kombinasjoner og kompleksitet - tips

- Lettere når enkeltdelene sitter
- Veksle mellom å fokusere på detaljer og å se helheten i kodeflyten

- Forstå kodeflyt.
  Følg programmet linje for linje f eks i Pythontutor og forutsi hva som skjer før du lar den utføre neste skritt.
  Bytt på å forklare for medstudenter.
- Mengdetrening bygger opp mentale modeller og forståelse

Fra underveisevaluering – hva er vanskelig?

Forstå oppgavetekst – tips

- Det hjelper når nye begreper sitter bedre (teori, mengdetrening, modning, ...)
- Skisser mulige løsninger – papir, kode, tegninger, ...
  => arbeid aktivt med teksten og ulike fremgangsmåter
- Hvis noe er tvetydig, prøv å identifisere konkrete alternative tolkninger
  - Hva er mest sannsynlig?
  - Hva klarer jeg å løse?
  - Løs så langt du klarer og gå så tilbake til teksten
  - Om fortsatt uklart:
    Skriv ned hva du velger og hvorfor (eksamen/ innlevering: Ta det med i besvarelsen)
- Husk: Å forstå problemet (oppgaven) er en sentral del av programmering!!
  - I Trix: Oppgaver med tag "problem"

PRAGMATISK PROGRAMMERING

Pragmatisk Programmering rider igjen!
Denne fredagen har vi fokus på å lese, forstå og løse oppgaver utifra det som faktisk står i oppgaveteksten.
Dette er høyst eksamensrelevant uansett hviket nivå du er på i kodingen.

I Pragmatisk Programmering bruker vi en muntlig tilnærming, leker og jobber sammen for å forstå. Kom og sjekk om det er noe for deg.



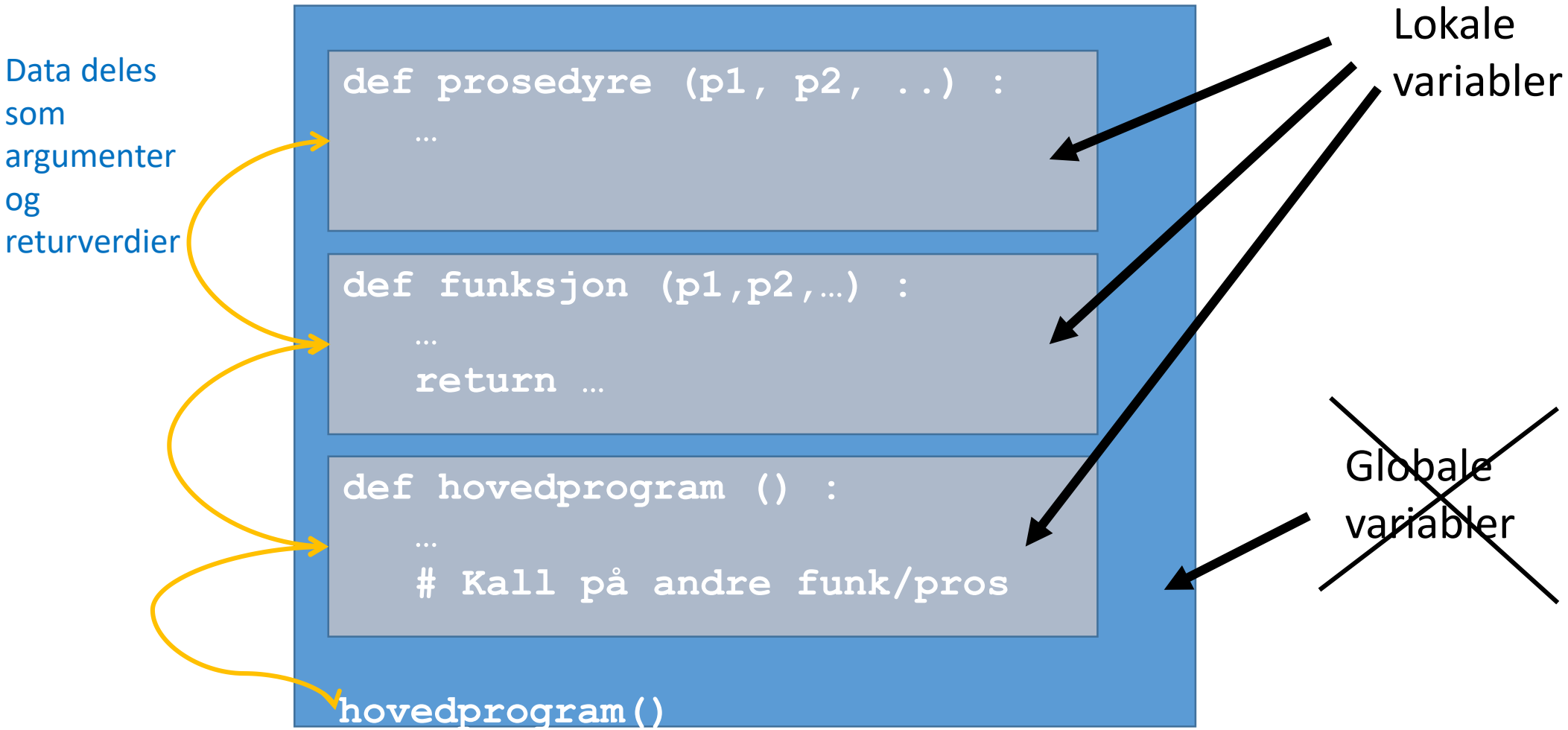
STED Smalltalk
DATO Fredag 06.10.23
TIDSPUNKT 16:15-18:00



Modularisering
Splitt & hersk

Lite tilbakeblikk: Programflyt og skop

mittProgram.py



Lite tilbakeblikk: Funksjonskall er uttrykk

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

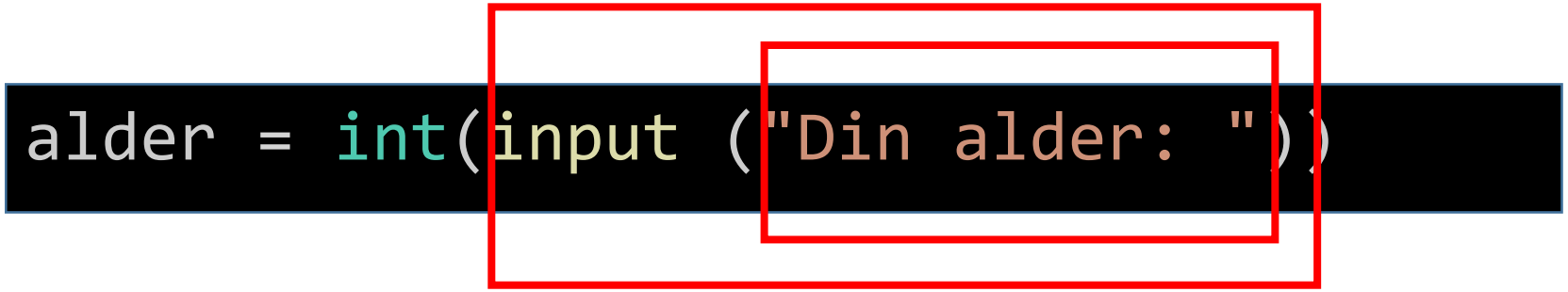
- Når vi trenger en verdi i et program har vi flere muligheter:
  - er det alltid den samme? Literal/ konstant
  - Har vi lagret verdien i en variabel?
  - Kan vi regne den ut?
  - Kan vi finne den ved å kalle på en funksjon?



Uttrykk

Lite tilbakeblikk: Funksjonskall er uttrykk

- Funksjonskall 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

- 1960 -> 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 (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) - 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(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^0_\mu(H\partial_\mu\phi^0 - \phi^0\partial_\mu H) + M(\frac{1}{c_w}Z^0_\mu\partial_\mu\phi^0 + W^+_\mu\partial_\mu\phi^- + W^-_\mu\partial_\mu\phi^+) - ig\frac{s_w^2}{c_w}MZ^0_\mu(W^+_\mu\phi^- - W^-_\mu\phi^+) + igs_w MA_\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_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^2}Z^0_\mu Z^0_\mu (H^2 + (\phi^0)^2 + 2(2s_w^2 - 1)^2\phi^+\phi^-) - \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^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{s_w}{c_w}(2c_w^2 - 1)Z^0_\mu A_\mu\phi^+\phi^- - g^2 s_w^2 A_\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_w A_\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) + \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_e}{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_\nu}{M}\phi^0(\bar{\nu}^\lambda\gamma^5\nu^\lambda) - \frac{ig}{2}\frac{m^\lambda_e}{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 på 50- og 60-tallet?

- Datamaskinene ble kraftigere
- billigere
- mer utbredt

=> Nye anvendelser!

- Helsevesenet
- Trafikk-planlegging
- Økonomisystemer

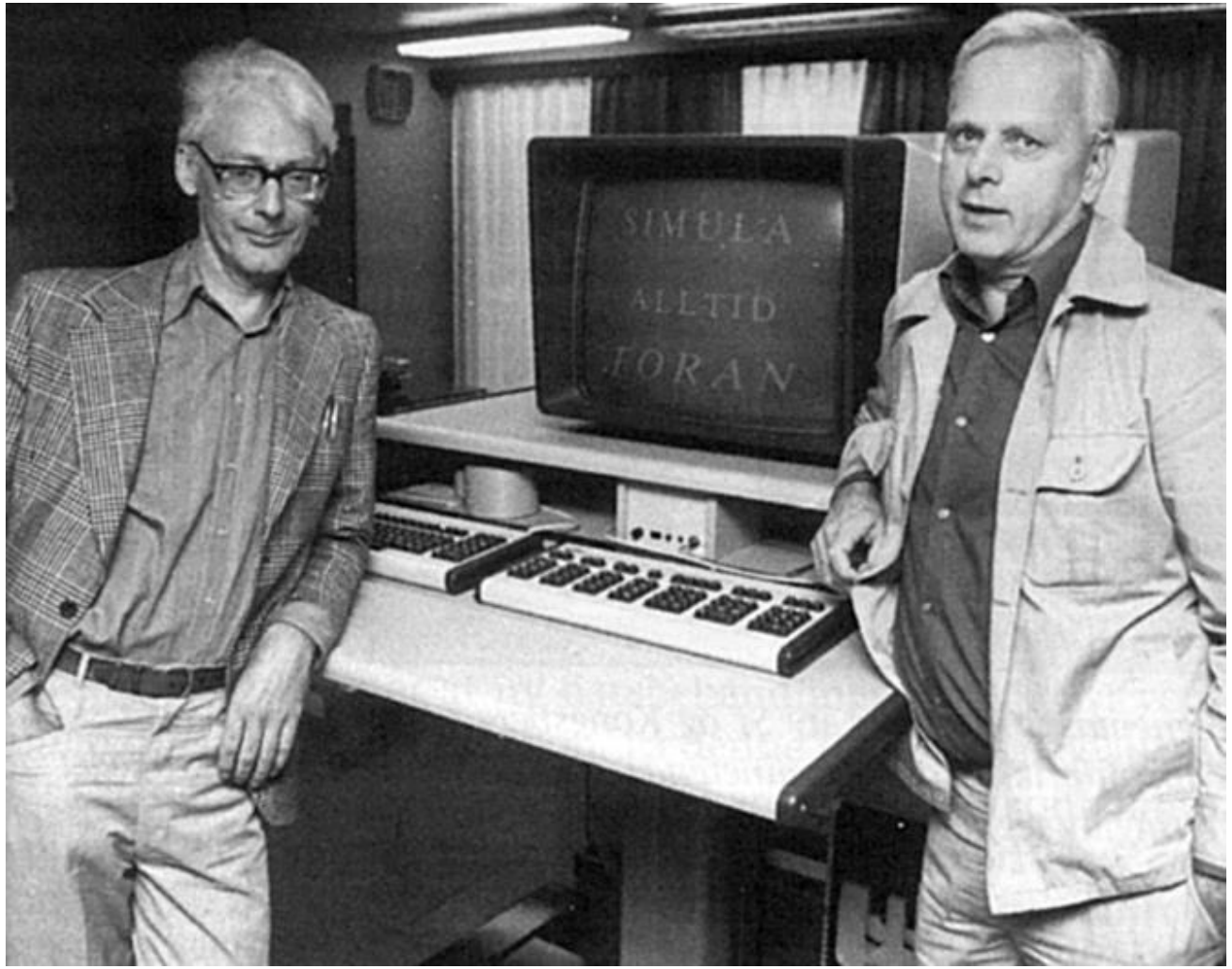
⇒ Nye tankesett, teori og programmeringsspråk for å håndtere størrelse og kompleksitet



Hvorfor er de to informatikk-byggene oppkalt etter Kristen Nygård og Ole-Johan Dahl?

Tildelt Turing-prisen 2001
(informatikkens Nobelpris)

.. for ideas fundamental to the emergence of object oriented programming, through their design of the programming languages Simula I and Simula 67



se f. eks. Wikipedia "Structured programming" og "Programming paradigms"

Programmeringsspråk og -paradigmer

- Maskinnære språk (maskinkode, assemby-kode, LMC)
  - Kan hoppe til "vilkårlige" steder i programmet/ minnet
- Strukturert programmering ble et begrep på 60-tallet
  - erstatter JUMP/ GOTO med konstruksjoner som beslutninger, løkker, subrutiner
    - unntak/ diskusjonstema: Tidlig return, exceptions, break/ continue
  - støttes av *imperative språk* som tilbyr
    - Prosedural programmering (det dere har gjort hittil) og/ eller
    - Objektorientert programmering (det vi starter med nå)
- Senere kom *deklarative språk*, f eks funksjonelle eller logiske språk

Objektorientert programmering: Kortere vei fra (kompleks) virkelighet til program

- Prosedyrer og funksjoner kan brukes for å løse deloppgaver

Modularisering
Splitt & hersk

- Hva om oppgavene handler om å bearbeide felles, kanskje komplekse, data?

- For eksempel trafikkregulering i et veikryss:
  - veier, veifelt, fotgjengeroverganger, trafikklys
  - endring og koordinering av lys basert på tidsintervaller, trykk på knapper, registrering av kjøretøy
  - Mange variabler, kompleks struktur, mange funksjoner => lett å miste oversikt

En måte å strukturere programmer på

- 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

Må fortsatt avgjøre: Hvilke data og operasjoner trengs? Hvordan gruppere i objekter?

Konseptet objektorientering (sitater fra Simula 50 år)

- 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

Et objekt representerer "noe"

Fysisk, konkret eller abstrakt

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

I et grafisk brukergrenssnitt

- et vindu
- en meny
- et felt i et vindu

Å programmere med objekter

Objekter er *instanser* av en klasse

Vi har tidligere brukt objekter som

- ..representerer lister m/ tilhørende tjenester

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

- ..representerer filer m/ tilhørende tjenester

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

Klasser

- Objekter har ulik oppførsel – tilbyr ulike tjenester avhengig av hvilken *klasse* de tilhører

- Klassedefinisjonen bestemmer
  - hvilke tjenester objektene tilbyr (grensesnittet)
  - hvordan de er implementert (metodedefinisjoner)
  - hvilke data objektet skal huske og hvordan de representeres (instansvariabler)

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"

```
liste = []
liste.append("ett_element")
```

- Vi utfører metoder på objektet vårt ved hjelp av kallet
  < objekt.metode(argumenter) >

- En metode kan ta argumenter (som en prosedyre/ funksjon)
- En metode kan returnere en verdi (som en funksjon)

Oppgave

- Finn et eller flere kodeeksempler der du har brukt "dot-notasjon" i et program.

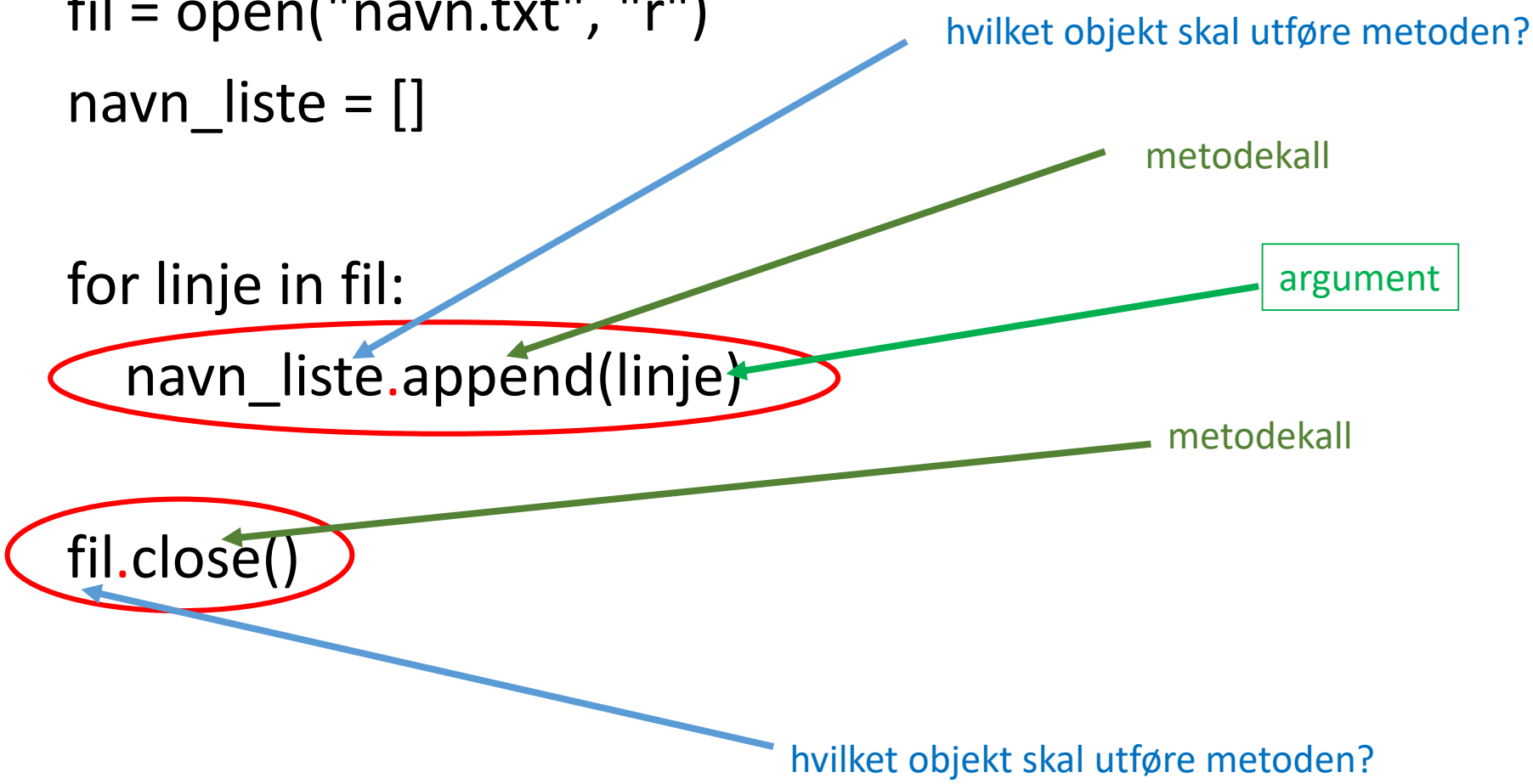
- For hvert eksempel: Identifiser
  - objektet
  - metodekall
  - argument(er)

Fra løsningsforslag Trix 5.01

```
fil = open("navn.txt", "r")
navn_liste = []

for linje in fil:
    navn_liste.append(linje)

fil.close()
```



Å lage egne klasser

Hva finnes bak grensesnittet?

Klasser

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

- Vi kan definere (programmere) våre egne klasser og opprette en eller flere instanser (objekter) av disse

Eksempel lyspære

- Lyspærer har ulike egenskaper – ulike systemer vil "se" dem på ulike måter
  - En lyspærefabrikk
  - Elektrobutikk
  - Smarthus-systemer

- Vi må lage en *modell* av en lyspære som (kun) omfatter *de egenskapene vi trenger til vårt bruk*

- Hva skal jeg bruke klassen til?

  => Styre lysene i et hus

Styring av smarte LED lyspærer

- Hva ønsker vi å kunne *gjøre* med lyspæren (versjon 0.1)?
  - Sjekke om den er på
  - Slå den på (tenne lyset)
  - Slå den av (slukke lyset)

Nå har vi (uformelt) beskrevet hvilke *tjenester* lyspære-objekter skal tilby – disse må implementeres som *metoder* i klassen.

En smart LED lyspære

{lyspære.py}

Klasse-definisjon (syntaks)

```
class Lyspære:

    def __init__(self):
        self._på = False

    def er_på(self):
        return self._på

    def tenne(self):
        self._på = True

    def slukke(self):
        self._på = False
```

klassenavn

konstruktør
m/ instansvariabel

(funksjons-)
metode

metode

metode

Klasse-definisjon

```
class Lyspære:

    def __init__(self):
        self._på = False

    def er_på(self):
        return self._på

    def tenne(self):
        self._på = True

    def slukke(self):
        self._på = False
```

- Alle objekter av en klasse tilbyr samme metoder
- Alle objekter av en klasse har (sine egne kopier av) samme instansvariabler
- Alle metodene i en klasse har tilgang til instansvariablene for et objekt

- Alle metoder må ha en parameter self, som husker hvilket objekt metoden ble kalt på
- Metodene bruker self for å aksessere instansvariablene i det objektet de er kalt for

Opprette objekter (syntaks)

- Noen innebygde klasser har snarveier for opprettelse av objekter
  - "" oppretter (og returnerer) en ny **`str`**
  - [] oppretter (og returnerer) en ny **`list`**

```
min_streng = ""

min_liste = []
```

- For de klassene vi selv definerer, opprettes (og returneres) objekter vha *klassenavn med argumentliste* (som kan være tom):

```
utelys = Lyspære()
```

Kall på metoder i et Lyspære-objekt (syntaks)

- Et objekt tilbyr bare de metodene som er definert i klassen

```
utelys = Lyspære()

status = utelys.er_på()

utelys.tenne()
```

Kaller på metoden **er_på()** som returnerer en boolsk verdi

Kaller på metoden **tenne()** som endrer verdien i instansvariabelen **_på** til **True**

Et testprogram for Lyspære

```
class Lyspære:

    def __init__(self):
        self._på = False

    def er_på(self):
        return self._på

    def tenne(self):
        self._på = True

    def slukke(self):
        self._på = False
```

```
# Testprogram for Lyspære

utelys = Lyspære()
assert not utelys.er_på()

utelys.tenne()
assert utelys.er_på()

utelys.slukke()
assert not utelys.er_på()

print("Alle tester gjennomført!")
```

Lyspære: Opprette et objekt av klassen

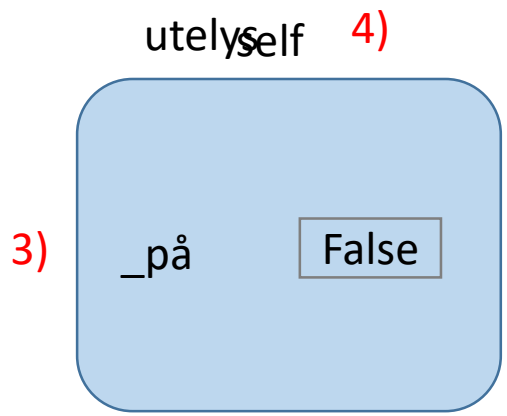
1. Klassenavnet gjør at det opprettes et nytt objekt
   Konstruktøren kalles automatisk når et nytt objekt opprettes:
2. **self** inneholder det nye objektet
3. oppretter og initialiserer instansvariablene i self-objektet
4. venstresiden får verdien til **self** (det nye objektet)

2)
3)

```
class Lyspære:
    def __init__(self):
        self._på = False

utelys = Lyspære()
```

4) 1)



Lyspære: Kalle en metode på et objekt

```
utelys = Lyspære()
entre = Lyspære()
bad = Lyspære()
stue = Lyspære()

utelys.tenne()
entre.tenne()
```

Metoden **tenne** utføres med **utelys** som argument for parameteren self!
Dvs variabelen før "." avgjør hvilken objekt metoden utføres for

```
def tenne(self):
    self._på = True
```

Mer om dette neste uke

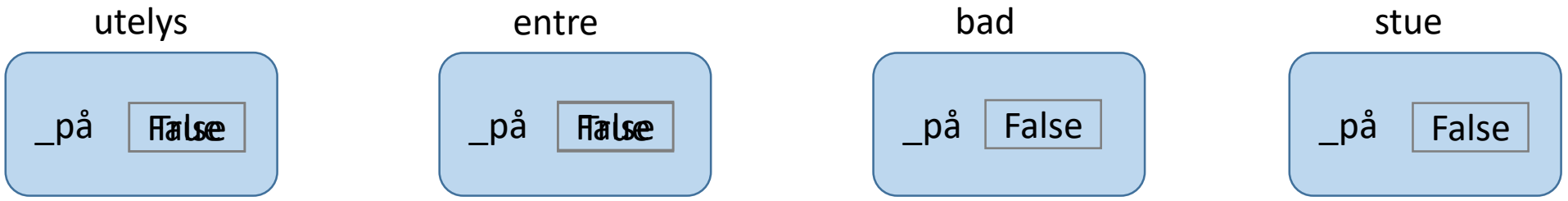
Instansvariabler og instansmetoder

- Hvert objekt som opprettes av en klasse får sine egne kopier av instansvariablene.
- Metodene har tilgang til instansvariablene i det objektet de kalles på

- Alle metodene i en klasse har tilgang til instansvariablene
- self-parameteren forteller metoden hvilket objekt den er kalt på
- alle instansvariabler må derfor aksesseseres med self i metodene

```
utelys = Lyspære()
entre = Lyspære()
bad = Lyspære()
stue = Lyspære()

utelys.tenne()
entre.tenne()
```



En ny klasse: Fremgangsmåte

Eksempel Student

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 type data og oppførsel (eks representere lyspærer eller studenter)
- For å simulere sammensatte fenomener vha objekter som representerer virkelige elementer (eks trafikk- simuleringer)

Fremgangsmåte objektorientert programmering

1. Identifiser aktuelle klasser (hva er sentrale «ting»/ begreper programmet skal behandle?)

2. Design klassens grensesnitt uformelt (hvilke tjenester ønsker jeg fra objekter av klassen?) og i Python (metodenavn/ parametere/ returverdier)
3. Design klassens datarepresentasjon (hvordan skal objektene representere dataene sine?)
4. Implementer metodene i Python (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 de bruker: Klassens grensesnitt ("public interface")

- Data og evt andre metoder (hjelpemetoder) definert inne i klassen er "non-public" og brukes kun inne i klassedefinisjonen (men der deles alt!)

- Sentralt prinsipp i objektorientert programmering
  - en klasse kan endres og utvides så lenge grensesnittet er det samme (eller bare med flere funksjoner) – uten at det får konsekvenser der klassen brukes
  - gjennomført ansvarsdeling (modularisering, splitt og hersk)

Innkapsling i Python og IN1000

- Python støtter innkapsling med *konvensjoner*, men har ikke mekanismer for å hindre tilgang
  - alle instansvariabler (og eventuelle 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* når og hvordan man anvender innkapsling
- Dere vil se eksempler på direkte aksess av data (utenom klassens metoder) blant annet i IN1010

Eksempel Student

1. Identifisere nyttig(e) klasse(r)
2. Design klassens grensesnitt
3. Design klassens datarepresentasjon
4. Implementer (fyll ut) metodene
5. Lag et testprogram

- 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?
  - Registrere poeng på innleveringer i IN1000 (eksempel fra fjorårets oblig-opplegg)

Student: Starter "utenfra"

1. Identifisere nyttig(e) klasse(r)
2. Design klassens grensesnitt
3. Design klassens datarepresentasjon
4. Implementer (fyll ut) metodene
5. Lag et testprogram

Design grensesnitt

- Ønsker å kunne gjøre følgende:
  - Lage et nytt Student-objekt med 0 poeng ved semesterstart
  - Øke studentens poeng med riktig antall når ny innlevering er rettet
  - Lese av (hente ut) antall poeng for en student

- Dvs: Grensesnittet må (i tillegg til konstruktør) tilby metoder for:
  - Registrerer nye poeng:
    `registrer_poeng(self, antall_poeng)  # endrer instansvariabel`
  - Henter antall poeng:
    `hent_poeng(self)  # Returnerer antall poeng studenten har`

Hvilke data må et objekt huske?

1. Identifisere nyttig(e) klasse(r)
2. Design klassens grensesnitt
3. Design klassens datarepresentasjon
4. Implementer (fyll ut) metodene
5. Lag et testprogram

- Antall poeng

  => vi trenger en instansvariabel som lagrer et heltall

Student – grensesnitt i Python
Oppgave: Fyll ut!

```
class Student:
    # Konstruktør, kun self-parameter
    # instansvariabel for poeng (int)
    def __init__(self):

    # Returnerer antall poeng
    def hent_poeng(self):

    # Øker poeng med antall
    def registrer_poeng(self, antall):
```

Student – implementasjon

```
class Student:
    # Konstruktør, kun self-parameter
    # instansvariabel for poeng (int)
    def __init__(self):
        self._poeng = 0

    # Returnerer antall poeng
    def hent_poeng(self):
        return self._poeng

    # Øker poeng med antall
    def registrer_poeng(self, antall):
        self._poeng += antall
```

Forbedring av studentklassen

- Klassen Student gir oss mulighet til å opprette så mange student-objekter vi vil
- Objektene vil ha hver sin instansvariabel som lagrer antall poeng for én student
- Men hvordan vet vi hvilket objekt som lagrer poengene til hvilken student?

1. Vi kan ha en variabel som forteller oss hvilken student hvert objekt tilhører
   ```
   ole = Student()
   dole = Student() # Upraktisk å måtte skrive hvert navn inn i programmet
   ```

2. Vi kan ha en ordbok som kobler navn med objekt
   ```
   studenter = {}
   studenter["ole"] = Student()
   studenter["dole"] = Student() # bedre, og vi kan lese inn navn til nøkler f eks fra fil
   ```

3. **Vi kan lagre studentenes navn i objektet sammen med poengsummen**
   (Vi kan fortsatt lagre objektene i en ordbok med navn som nøkkel. Mer om dette neste gang)
4. **Vi vet navnet på studenten når vi oppretter et nytt objekt – dette kan vi oppgi til konstruktøren**

Oppgave:
Ny klasse StudentMedNavn

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

    def hent_poeng(self):
        return self._poeng

    def registrer_poeng(self, antall):
        self._poeng += antall
```

Sammenlignet med Student-klassen til venstre skal du:

1. Endre konstruktør:
   - Ny parameter navn
   - Ny instansvariabel _navn

2. Ny metode for å skrive ut navn og poeng på terminal
   - ny metode skriv_ut

3. Hvordan opprettes et objekt av klassen StudentMedNavn?

Oppgave

```
class StudentMedNavn:
    def __init__(
        self._poeng = 0


    def hent_poeng(self):
        return self._poeng

    def registrer_poeng(self, antall):
        self._poeng += antall

    def skriv_ut(
```

Sammenlignet med Student-klassen til venstre skal du:

1. Endre konstruktør:
   - Ny parameter navn
   - Ny instansvariabel _navn
2. Ny metode for å skrive ut navn og poeng på terminal
   - ny metode skriv_ut
3. Hvordan opprettes et objekt av klassen StudentMedNavn?

Oppgave - løsningsforslag

```
class StudentMedNavn:
    def __init__(self, navn):
        self._poeng = 0
        self._navn = navn

    def hent_poeng(self):
        return self._poeng

    def registrer_poeng(self, antall):
        self._poeng += antall

    def skriv_ut(self):
        print(self._navn, "har totalt", self._poeng, "poeng")
```

Sammenlignet med Student-klassen til venstre skal du:

1. Endre konstruktør:
   - Ny parameter navn
   - Ny instansvariabel _navn
2. Ny metode for å skrive ut navn og poeng på terminal
   - ny metode skriv_ut
3. Hvordan opprettes et objekt av klassen StudentMedNavn?

Test av klassen (slik dere skal levere obliger)

- Vi skriver et testprogram i en egen fil
- Importerer klassen StudentMedNavn (fra filen *student_med_navn.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 - og forstyrrer ikke kjøringen når alt er ok

Testprogram for StudentMedNavn-klassen

```
from student_med_navn import StudentMedNavn

def hovedprogram():
    # tester klassen Student_med_navn
    stud = StudentMedNavn("Einstein")

    stud.skriv_ut()                 # Er innholdet som forventet og fungerer skriv_ut?
    assert stud.hent_poeng() == 0   # Ble objektet opprettet med forventet poengsum?

    stud.registrer_poeng(5)
    assert stud.hent_poeng() == 5   # Ble antall poeng oppdatert som forventet?
    stud.skriv_ut()                 # Er innholdet fortsatt som forventet i skriv_ut?

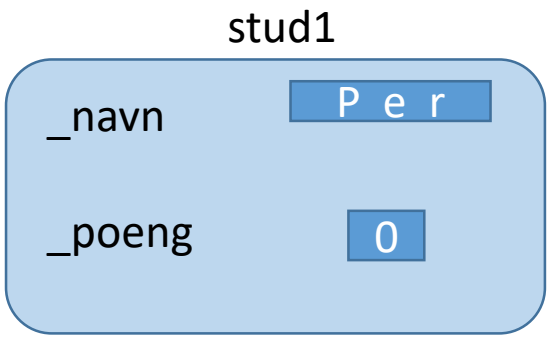
hovedprogram()
```

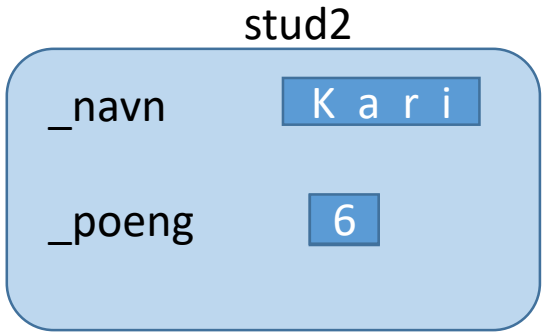
Studentobjekter

- 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 = StudentMedNavn("Per")
stud2 = StudentMedNavn("Kari")
stud2.registrer_poeng(6)
```





test_flere_studenter.py

Lite eksempel: liste med flere objekter

```
from student_med_navn import StudentMedNavn

studenter_liste = []
studenter_liste.append(StudentMedNavn("Ole"))
studenter_liste.append(StudentMedNavn("Dole"))
studenter_liste.append(StudentMedNavn("Doffen"))

studenter_liste[0].registrer_poeng(3)
studenter_liste[0].registrer_poeng(5)
studenter_liste[0].registrer_poeng(5)
studenter_liste[0].registrer_poeng(6)

print("Studenter med nok poeng (fra liste):")
for student in studenter_liste:
    if student.hent_poeng() >= 19:
        student.skriv_ut()
```

Utskrift:

```
Studenter med nok poeng (fra liste):
Ole har totalt 19 poeng
```

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 (Jonas Gahr Støre)

Klassen Navn - grensesnitt

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

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

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

Klassen Navn - implementasjon

Filen navn.py

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

  def sortert(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
```

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
```

Oppsummering

(se også notatet lenket fra uke 7-siden for 2 sider tekst)

Læringsmål uke 7

- Kjenne til motivasjon og bakgrunn for objektorientert programmering
- Programmering: Kunne definere en klasse, opprette objekter og kalle metoder på objektene
- Kunne bruke og forklare begrepene klasse, objekt, klassedefinisjon, konstruktør, metode og instansvariabel
- Forstå sentrale begreper som grensesnitt og innkapsling
- Kjenne til utviklingsprosessen for en klasse gjennom design, implementasjon og testing

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 metoder

```
class Lyspære:

    def __init__(self):
        self._på = False

    def er_på(self):
        return self._på

    def tenne(self):
        self._på = True

    def slukke(self):
        self._på = False
```

klassenavn

konstruktør m/ instansvariabel

(funksjons-) metode

metode

metode

```
# Testprogram for Lyspære

utelys = Lyspære()
assert not utelys.er_på()

utelys.tenne()
assert utelys.er_på()

utelys.slukke()
assert not utelys.er_på()

print("Alle tester gjennomført!")
```

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!