

Introduksjon til klasser (kapittel 7)

Joakim Sundnes^{1,2}

Simula Research Laboratory¹

Universitetet i Oslo, Institutt for informatikk²

Oct 13, 2026

Tirsdag 13/10:

- Oppgave 5.16, 5.18 (numpy, plotting, lesing fra fil, moduler)
- Oppgave 6.9 (Dictionaries, strenger)
- Introduksjon til klasser

Torsdag 15/10:

- Oppgave 7.1, 7.2
- Mer om klasser:
 - Beskyttede attributter
 - Spesialmetoder
- Oppgave 7.10, 7.11, (+ kanskje 6.11)

Grunnleggende om klasser (1)

- Klasser er en essensiell del av *objektorientert programmering*
- Vi har brukt klasser siden dag 1 i IN1900:

```
>>> S = "Dette er en streng"
>>> type(S)
>>> <class 'str'>
>>> L = S.split()
>>> type(L)
>>> <class 'list'>
```

Grunnleggende om klasser (2)

- Klasser pakker sammen data og funksjoner som naturlig hører sammen
- Hver gang vi lager et streng-objekt i Python, lager vi en *instans* av den innebygde klassen `str`
- Kall som `S.split()` kaller funksjonen `split()` som tilhører instansen `S`
- Vi skal nå lære å lage våre egne klasser

Nye termer og definisjoner

- **Klasse:** Definisjon av en ny datatype som inneholder data og funksjoner som naturlig hører sammen.
- **Instans:** Et objekt laget fra en klasse. Etter å ha definert en klasse, kan vi lage mange instanser av den.
- **Metode:** En funksjon som er bundet til en instans av en klasse.
- **Attributt:** En variabel som er bundet til en instans av en klasse.
- En klasse består av *attributter* og *metoder*.

Klasse = funksjoner + data (variabler) i én enhet

- En klasse pakker sammen data (attributter) og funksjoner (metoder) som *én enkelt enhet*
- Som programmerer kan du lage en ny klasse og dermed en ny objekttype (som float, list, file, ...)
- En klasse ligner litt på en modul: en samling av "globale" variabler og funksjoner som hører sammen
- Det finnes bare én instans av en modul, mens en klasse kan ha mange instanser (kopier)
- Klasser brukes mye i moderne programmering
- Eksempler er nyttig for å introdusere klassekonseptet:
 - (Klasser som representerer fysiske objekter (avsnitt 7.2.1 i læreboka))
 - Klasser for matematiske funksjoner
 - Vår egen klasse for tekststrenger

Representere en funksjon med en klasse; bakgrunn (1)

Tenk på en funksjon av t med en parameter v_0 :

$$y(t; v_0) = v_0 t - \frac{1}{2} g t^2$$

Vi trenger både v_0 og t for å evaluere y (og $g = 9.81$), men hvordan bør vi implementere dette?

Å ha t og v_0 som argumenter?

```
def y(t, v0):  
    g = 9.81  
    return v0 * t - 0.5 * g * t**2
```

Representere en funksjon med en klasse; bakgrunn (2)

Implementeringen med to argumenter fungerer vanligvis, men den er ikke alltid praktisk.

Hva hvis funksjonen skal sendes som et argument til en annen funksjon som forventer at den tar nøyaktig ett argument? (Veldig vanlig situasjon i Python-programmer)

- Eksempel: Newtons metode fra uke 8 (kap 11.2 i læreboka):
 - Funksjonen `Newton(f, dfdx, x0, tol)` forventer at det første argumentet er en Python-funksjon som tar ett enkelt argument som input.
 - Funksjonen vi feile hvis vi sender med vår `y(t, v0)`-funksjon som argument.

Mulige (dårlige) løsninger

Å ha t som argument og v_0 som global variabel?

```
def y(t):  
    g = 9.81  
    return v0 * t - 0.5 * g*t**2
```

Problem: Fungerer bare hvis v_0 er definert som en global variabel før funksjonen brukes (veldig dårlig løsning).

Å ha t som argument og v_0 som lokal variabel?

```
def y(t):  
    v0 = 3  
    g = 9.81  
    return v0 * t - 0.5 * g * t**2
```

Problem: Hva hvis vi ønsker en annen verdi for v_0 ? Da må vi redigere funksjonen. (Litt bedre løsning enn den forrige, men fortsatt ikke god.)

$$y(t; v_0) = v_0 t - \frac{1}{2} g t^2$$

- Husk at en klasse pakker sammen variabler og funksjoner som naturlig hører sammen
- Vi kan lage en klasse som inneholder v_0 og g som variabler (attributter), og en funksjon (metode) $y(t)$
- Med en klasse kan $y(t)$ være en funksjon kun av t , men likevel ha v_0 og g som parametere med gitte verdier.
- Klassen pakker sammen en funksjon (metode) $y(t)$ og data (attributter) v_0 , g .

Representere en funksjon med en klasse; implementasjon

- Vi lager en klasse Y for $y(t; v_0)$ med variablene v_0 og g og en funksjon `value(t)` for å beregne $y(t; v_0)$
- Klassen trenger en metode `__init__` (konstruktør) for initialisering av variablene
- Selve funksjonsuttrykket kan skrives som en egen metode i klassen
- Når vi har skrevet koden for klassen kan vi lage *instanser* av den, med forskjellige verdier for v_0

Representere en funksjon med en klasse; koden

```
class Y:
    def __init__(self, v0):
        self.v0 = v0
        self.g = 9.81

    def value(self, t):
        return self.v0*t - 0.5*self.g*t**2
```

Bruk:

```
y = Y(v0 = 3)           # lag instans (objekt)
height = y.value(0.1)    # beregn funksjonsverdi
```

Representere en funksjon med en klasse; konstruktøren

Når vi skriver

```
y = Y(v0=3)
```

lager vi en ny variabel (instans) *y* av klassen *Y*.

Uttrykket *Y(v0=3)* vil automatisk kalle på *konstruktøren* til *Y*:

```
def __init__(self, v0):  
    self.v0 = v0  
    self.g = 9.81
```

Hva er denne self-variabelen?

Forvirrende i starten, men man blir vant til det etter hvert.

- Tenk på `self` som `y`, altså den nye variabelen (attributten) som skal lages. `self.v0 = ...` betyr at vi knytter et attributt `v0` til `self` (dvs. til `y`).
- `Y(3)` betyr `Y.__init__(y, 3)`, altså kall konstruktøren med `self=y, v0=3`
- Husk: `self` er alltid den første parameteren i implementeringen av en metode, men blir aldri inkludert som argument når vi kaller på metoden!
- Etter `y = Y(3)` har `y` to attributter, `v0` og `g`

```
print(y.v0)
print(y.g)
```

"In mathematics you don't understand things. You just get used to them." – John von Neumann

Hva er denne self-variabelen?

Forvirrende i starten, men man blir vant til det etter hvert.

- Tenk på `self` som `y`, altså den nye variabelen (attributten) som skal lages. `self.v0 = ...` betyr at vi knytter et attributt `v0` til `self` (dvs. til `y`).
- `Y(3)` betyr `Y.__init__(y, 3)`, altså kall konstruktøren med `self=y, v0=3`
- Husk: `self` er alltid den første parameteren i implementeringen av en metode, men blir aldri inkludert som argument når vi kaller på metoden!
- Etter `y = Y(3)` har `y` to attributter, `v0` og `g`

```
print(y.v0)
print(y.g)
```

"In mathematics you don't understand things. You just get used to them." – John von Neumann

Representere en funksjon med en klasse; value-metoden

Implementasjonen av value-metoden:

```
def value(self, t):  
    return self.v0*t - 0.5*self.g*t**2
```

Eksempel på et kall:

```
v = y.value(t=0.1)
```

Vi inkluderer ikke `self` i kallet, men Python setter automatisk inn `y` som `self`-argumentet inne i `value`-metoden. Tenk på dette som

```
Y.value(y, t=0.1)
```

`self` gir tilgang til attributtene til instansen `y`.

Flere detaljer i Avsnitt 7.2.1-7.2.2 i læreboka.

Skisse av en generell Python-klasse

```
class MinKlasse:
    def __init__(self, p1, p2):
        self.attr1 = p1
        self.attr2 = p2

    def metode1(self, arg):
        return self.attr1 + self.attr2 + arg

    def metode2(self):
        print('Hallo!')

m = MinKlasse(4, 10)
print(m.metode1(-2))
m.metode2()
```

Eksempel 2: Hva hvis datatypen 'str' ikke fantes?

- Vi har sett hvor nyttig det er å ha funksjoner som `split` og `strip` når vi jobber med tekststrenger
- La oss anta at datatypen `str` ikke hadde disse funksjonene. Kan vi lage vår egen datatype som har dem?
 - Vi må lage en *klasse*, som har selve teksten som *attributt* og de funksjonene vi ønsker som *metoder*

Vår egen (enkle) tekststreng-klasse

```
class Tekst:
    def __init__(self, innhold):
        self.innhold = innhold

    def split(self, arg = ' '):
        res = []
        word = ''
        for e in self.innhold:
            if e != arg:
                word += e
            elif len(word) > 0:
                res.append(word)
                word = ''
        res.append(word)
        return res
```

Oppsummering av klasseintroduksjon

- En klasse er rett og slett en samling av funksjoner og data som naturlig hører sammen
- Funksjoner i en klasse kalles vanligvis metoder, data kalles attributter
- Vi lager instanser (eller objekter) av en klasse, og hver instans kan ha forskjellige verdier for attributtene
- Alle klasser bør ha en metode `__init__`, kalt en konstruktør, som kalles hver gang en ny instans lages
- Konstruktøren vil typisk initialisere all data i en instans
- Alle metoder i en klasse skal ha `self` som første argument i definisjonen, men ikke i kallet. Dette kan være forvirrende i starten, men man blir vant til det.

Hvorfor bruke klasser (1)?

- For korte, enkle Python-programmer er klasser egentlig aldri nødvendig, men de kan gjøre et program ryddigere og mer lesbart
- For store og komplekse programmer er ryddig og lesbar kode ekstremt viktig
- Viktigere i andre programmeringsspråk (Java, C++, etc)
- Python har praktiske innebygde datatyper (lister, dictionaries) som gjør det mindre viktig å lage egne klasser
- Klasser og objektorientert programmering (OOP) er standardverktøy i programvareutvikling
- OOP ble oppfunnet ved Universitetet i Oslo (!)

Hvorfor bruke klasser (2)

Tenk på hvordan vi har brukt str-klassen:

```
>>> a = "dette er en streng"
>>> type(a)
>>> <class 'str'>
>>> l = a.split()
```

Python-utviklerne kunne ha løst dette uten klasser, ved å gjøre `split` til en global funksjon:

```
>>> a = "dette er en streng"
>>> l = split(a)
```

(Advarsel: dette virker ikke, det er bare et eksempel på hvordan ting *kunne* vært gjort.)

Fordelen med klasseløsningen er at den pakker sammen data og funksjoner som naturlig hører sammen.

Et annet klasseeksempel: en bankkonto

- **Attributter:** navnet på eieren, kontonummer, saldo
- **Metoder:** innskudd, uttak, penutskrift

```
class Account:
    def __init__(self, name, account_number, initial_amount):
        self.name = name
        self.no = account_number
        self.balance = initial_amount

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def dump(self):
        s = f'{self.name}, {self.no}, balance: {self.balance}'
        print(s)
```

Eksempel på bruk av klassen Konto

```
>>> a1 = Account('John Olsson', '19371554951', 20000)
>>> a2 = Account('Liz Olsson', '19371564761', 20000)
>>> a1.deposit(1000)
>>> a1.withdraw(4000)
>>> a2.withdraw(10500)
>>> a1.withdraw(3500)
>>> print("a1's balance:", a1.balance)
a1's balance: 13500
>>> a1.dump()
John Olsson, 19371554951, balance: 13500
>>> a2.dump()
Liz Olsson, 19371564761, balance: 9500
```

Hvordan klassen *ikke* er ment å brukes

Mulig, men ikke tiltenkt bruk:

```
>>> a1.name = 'Some other name'  
>>> a1.balance = 100000  
>>> a1.no = '19371564768'
```

Riktig bruk av klassen:

- Attributtene skal *ikke* endres direkte!
- balance-attributten kan ses på
- Endring av balance gjøres gjennom metodene withdraw eller deposit

Løsning:

Attributter og metoder som ikke er ment for bruk utenfor klassen, kan merkes som *beskyttet* ved å sette en understrek foran navnet (f.eks. `_navn`). Dette er bare en konvensjon. Det hindrer ingen i å bruke metodene og attributtene direkte, men det forteller andre at det ikke er ment å brukes på denne måten.

Hvordan klassen *ikke* er ment å brukes

Mulig, men ikke tiltenkt bruk:

```
>>> a1.name = 'Some other name'  
>>> a1.balance = 100000  
>>> a1.no = '19371564768'
```

Riktig bruk av klassen:

- Attributtene skal *ikke* endres direkte!
- balance-attributten kan ses på
- Endring av balance gjøres gjennom metodene withdraw eller deposit

Løsning:

Attributter og metoder som ikke er ment for bruk utenfor klassen, kan merkes som *beskyttet* ved å sette en understrek foran navnet (f.eks. `_navn`). Dette er bare en konvensjon. Det hindrer ingen i å bruke metodene og attributtene direkte, men det forteller andre at det ikke er ment å brukes på denne måten.

Hvordan klassen *ikke* er ment å brukes

Mulig, men ikke tiltenkt bruk:

```
>>> a1.name = 'Some other name'  
>>> a1.balance = 100000  
>>> a1.no = '19371564768'
```

Riktig bruk av klassen:

- Attributtene skal *ikke* endres direkte!
- balance-attributten kan ses på
- Endring av balance gjøres gjennom metodene withdraw eller deposit

Løsning:

Attributter og metoder som ikke er ment for bruk utenfor klassen, kan merkes som *beskyttet* ved å sette en understrek foran navnet (f.eks. `_navn`). Dette er bare en konvensjon. Det hindrer ingen i å bruke metodene og attributtene direkte, men det forteller andre at det ikke er ment å brukes på denne måten.

Hvordan klassen *ikke* er ment å brukes

Mulig, men ikke tiltenkt bruk:

```
>>> a1.name = 'Some other name'  
>>> a1.balance = 100000  
>>> a1.no = '19371564768'
```

Riktig bruk av klassen:

- Attributtene skal *ikke* endres direkte!
- balance-attributten kan ses på
- Endring av balance gjøres gjennom metodene withdraw eller deposit

Løsning:

Attributter og metoder som ikke er ment for bruk utenfor klassen, kan merkes som *beskyttet* ved å sette en understrek foran navnet (f.eks. `_navn`). Dette er bare en konvensjon. Det hindrer ingen i å bruke metodene og attributtene direkte, men det forteller andre at det ikke er ment å brukes på denne måten.

Forbedret klasse med beskyttede attributter

```
class AccountP:
    def __init__(self, name, account_number, initial_amount):
        self._name = name
        self._no = account_number
        self._balance = initial_amount

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        self._balance -= amount

    def get_balance(self):      # NEW - read balance value
        return self._balance

    def dump(self):
        s = f'{self._name}, {self._no}, balance: {self._balance}'
        print(s)
```

Bruk av den nye klassen KontoP

```
a1 = AccountP('John Olsson', '19371554951', 20000)
a1.withdraw(4000)

print(a1._balance)      # it works, but a convention is broken
print(a1.get_balance()) # correct way of viewing the balance
a1._no = '19371554955' # also works, but is a "serious crime"!
```

Spørsmål: Hvorfor er dette nyttig?

Hint: Tenk på store bibliotekskoder som skal brukes av mange andre programmerere i mange år.