

Objektorientert programmering (kap 7.4)

Joakim Sundnes^{1,2}

Simula Research Laboratory¹

Universitetet i Oslo, Institutt for informatikk²

Oct 20, 2026

Tirsdag 20. okt:

- Oppgave 7.3, 7.12, 7.25 fra ("A primer on...")
- Introduksjon til objektorientert programmering (OOP)

Torsdag 22. okt:

- Oppgave 9.1, 9.3, 9.4, 9.6 (fra "A primer on...")
- Mer om objektorientert programmering:
 - Klasser for numerisk derivasjon

Quiz: Grunnleggende om klasser

```
from math import sin

class Fnc:
    def __init__(self, v0, v1):
        self.v0 = v0
        self.v1 = v1

    def __call__(self, x):
        return self.v0*sin(self.v1*x)

    def __str__(self):
        return f'f(x) = {self.v0} * sin({self.v1}*x) '

    def df(self, x):
        return self.v0*self.v1*cos(self.v1*x)
```

Spørsmål:

- Hvor mange data-attributter har klassen?
- Hvor mange metoder har klassen?
- Hvorfor har noen metoder navn på formen `__xxx__`?
- Hva er variabelen `self`?

Quiz: Grunnleggende om klasser

```
from math import sin

class Fnc:
    def __init__(self, v0, v1):
        self.v0 = v0
        self.v1 = v1

    def __call__(self, x):
        return self.v0*sin(self.v1*x)

    def __str__(self):
        return f'f(x) = {self.v0} * sin({self.v1}*x)'

    def df(self, x):
        return self.v0*self.v1*cos(self.v1*x)
```

Spørsmål:

- Hvilke metoder blir kalt av kodelinjene nedenfor?

```
f = Fnc(7,2)
print(f)
print(f(3.5))
print(f.df(3.5))
```

Tittelen *Objektorientert programmering* (OOP) kan bety to forskjellige ting

- 1 Programmering med klasser og objekter (kalles også *objektbasert* programmering)
- 2 **Programmering med klassehierarkier** (familier av klasser)

Nytt konsept: samle klasser i familier (hierarkier)

Hva er et klassehierarki?

- En familie av nært beslektede klasser
- Et nøkkelkonsept er *arv*: subklasser arver attributter og metoder fra superklasser/baseklasser, som minimerer duplisering av kode og sparer mye skriving.

OOP ble oppfunnet av Ole-Johan Dahl og Kristen Nygaard på 1960-tallet - en viktig oppfinnelse i informatikk, som brukes i alle store datasystemer i dag.

Objektorientert programmering

- Alt i Python er et objekt, så all Python-programmering er *objektbasert*
- Objektorientert programmering (OOP) tar ideene om klasser et skritt videre
- Vi utnytter og utvider en veldig nyttig egenskap ved klasser; at de kan kombineres og gjenbrukes som byggeklosser.
- Hvis vi definerer en klasse `class A`, kan vi definere en annen klasse `class B(A)`.
- Klasse B *arver* alle attributter og metoder fra A
- Klassen blir en *utvidelse* av klasse A
- Vi sier at A er en superklasse eller baseklasse, og B er en subklasse av A

```
class A:
    def __init__(self, v0, v1):
        self.v0 = v0
        self.v1 = v1

    def f(self, x):
        return x**2

class B(A):
    def g(self, x):
        return x**4

class C(B):
    def h(self, x):
        return x**6
```

Vi har nå definert tre klasser:

- A: to attributter (v0, v1) og to metoder (__init__, f)
- B: to attributter (v0, v1) og tre metoder (__init__, f, g)
- C: to attributter (v0, v1) og fire metoder (__init__, f, g, h)

OOP er enda viktigere i andre programmeringsspråk

Språk som Java og C++ har statisk typing. En funksjon blir deklartert til å ta inn argumenter av en bestemt type:

```
void my_func(A my_obj)
{
    ...
}
```

OOP gir ekstra fleksibilitet, siden denne funksjonen også vil akseptere argumenter av klassene B og C.

OOP er veldig nyttig også i Python, for å unngå kodeduplisering og produsere strukturert og lesbar kode!

Grunnleggende OOP - overstyring av metoder

En subklasse kan overstyre metoder i superklassen. Si at vi ønsker å legge til noen ekstra attributter i en subklasse:

```
class A:
    def __init__(self, v0, v1):
        self.v0 = v0
        self.v1 = v1

    def f(self, x):
        return x**2

class B(A):
    def __init__(self, v0, v1, v2):
        self.v0 = v0
        self.v1 = v1
        self.v2 = v2

    def g(self, x):
        return x**4
```

Bruk:

```
a = A(v0=1, v1=2).      #kaller A.__init__
b = B(v0=1, v1=2, v3=3) #kaller B.__init__
```

Den overstyrte metoden kan fortsatt kalles

En mer elegant implementasjon ser slik ut:

```
class A:
    def __init__(self, v0, v1):
        self.v0 = v0
        self.v1 = v1

    def f(self, x):
        return x**2

class B(A):
    def __init__(self, v0, v1, v2):
        A.__init__(v0, v1) #eller super().__init__(self, v0, v1)
        self.v2 = v2

    def g(self, x):
        return x**4
```

Eksempel 1: klasser for lineære og kvadratiske funksjoner

Problem:

Lag en klasse for å representere lineære funksjoner $y = c_0 + c_1x$.

Kode:

```
import numpy as np

class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1

    def __call__(self, x):
        return self.c0 + self.c1*x

    def table(self, L, R, n):
        """Returner en tabell med n punkter for  $L \leq x \leq R$ ."""
        s = ''
        for x in np.linspace(L, R, n):
            y = self(x)
            s += f'{x:12g} {y:12g}\n'
        return s
```

Eksempel 1: klasser for lineære og kvadratiske funksjoner

Problem:

Lag en klasse for å representere lineære funksjoner $y = c_0 + c_1x$.

Kode:

```
import numpy as np

class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1

    def __call__(self, x):
        return self.c0 + self.c1*x

    def table(self, L, R, n):
        """Returner en tabell med n punkter for  $L \leq x \leq R$ ."""
        s = ''
        for x in np.linspace(L, R, n):
            y = self(x)
            s += f'{x:12g} {y:12g}\n'
        return s
```

Eksempel 1: klasser for lineære og kvadratiske funksjoner

Problem:

Lag en klasse for å representere lineære funksjoner $y = c_0 + c_1x$.

Kode:

```
import numpy as np

class Line:
    def __init__(self, c0, c1):
        self.c0, self.c1 = c0, c1

    def __call__(self, x):
        return self.c0 + self.c1*x

    def table(self, L, R, n):
        """Returner en tabell med n punkter for L <= x <= R."""
        s = ''
        for x in np.linspace(L, R, n):
            y = self(x)
            s += f'{x:12g} {y:12g}\n'
        return s
```

En klasse for parabler

Problem:

Lag en klasse for å evaluere kvadratiske funksjoner (parabler):

$$y = c_0 + c_1x + c_2x^2.$$

Kode:

```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0, self.c1, self.c2 = c0, c1, c2

    def __call__(self, x):
        return self.c2*x**2 + self.c1*x + self.c0

    def table(self, L, R, n):
        """Returner en tabell med n punkter for L <= x <= R."""
        s = ''
        for x in np.linspace(L, R, n):
            y = self(x)
            s += f'{x:12g} {y:12g}\n'
        return s
```

Dette er nesten den samme koden som klassen Line, bortsett fra leddet med c2

En klasse for parabler

Problem:

Lag en klasse for å evaluere kvadratiske funksjoner (parabler):

$$y = c_0 + c_1x + c_2x^2.$$

Kode:

```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0, self.c1, self.c2 = c0, c1, c2

    def __call__(self, x):
        return self.c2*x**2 + self.c1*x + self.c0

    def table(self, L, R, n):
        """Returner en tabell med n punkter for L <= x <= R."""
        s = ''
        for x in np.linspace(L, R, n):
            y = self(x)
            s += f'{x:12g} {y:12g}\n'
        return s
```

Dette er nesten den samme koden som klassen Line, bortsett fra leddet med c_2

En klasse for parabler

Problem:

Lag en klasse for å evaluere kvadratiske funksjoner (parabler):

$$y = c_0 + c_1x + c_2x^2.$$

Kode:

```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0, self.c1, self.c2 = c0, c1, c2

    def __call__(self, x):
        return self.c2*x**2 + self.c1*x + self.c0

    def table(self, L, R, n):
        """Returner en tabell med n punkter for L <= x <= R."""
        s = ''
        for x in np.linspace(L, R, n):
            y = self(x)
            s += f'{x:12g} {y:12g}\n'
        return s
```

Dette er nesten den samme koden som klassen Line, bortsett fra leddet med c_2

En klasse for parabler

Problem:

Lag en klasse for å evaluere kvadratiske funksjoner (parabler):

$$y = c_0 + c_1x + c_2x^2.$$

Kode:

```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0, self.c1, self.c2 = c0, c1, c2

    def __call__(self, x):
        return self.c2*x**2 + self.c1*x + self.c0

    def table(self, L, R, n):
        """Returner en tabell med n punkter for L <= x <= R."""
        s = ''
        for x in np.linspace(L, R, n):
            y = self(x)
            s += f'{x:12g} {y:12g}\n'
        return s
```

Dette er nesten den samme koden som klassen Line, bortsett fra leddet med c2

Klassen Parabola som subklasse av Line; prinsipper

- Parabola-kode = Line-kode + litt ekstra med c_2 -leddet
- Kan vi utnytte Line-kode i klassen Parabola?
- Dette er det arv handler om!

Ved å skrive

```
class Parabola(Line):  
    pass
```

gjør at Parabola arver alle metoder og attributter fra Line, så Parabola har attributtene c_0 og c_1 og tre metoder

- Line er en *superklasse*, Parabola er en *subklasse* <linebreak> (forelderklasse, baseklasse; barneklasse, avledet klasse)
- Klassen Parabola må legge til kode i Line sin konstruktør (en ekstra c_2 -attributt), `__call__` (ett ekstra ledd), men table kan brukes uendret
- Ideen er å gjenbruke så mye kode i Line som mulig og unngå å duplisere kode

Klasse Parabola som subklasse av Line; kode

En subklasse-metode kan kalle en superklasse-metode med
`superklasse_navn.metode(self, arg1, arg2, ...)`

eller med den innebygde funksjonen `super()`:

`super().metode(self, arg1, arg2, ...)`

Klassen Parabola som en subklasse av Line:

```
class Parabola(Line):
    def __init__(self, c0, c1, c2):
        super().__init__(self, c0, c1)  # Line lagrer c0, c1
        self.c2 = c2

    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2
```

Hva er poenget?

- Klassen Parabola legger bare til kode i den eksisterende koden i klassen Line - ingen duplisering av lagring av c_0 og c_1 , og beregning av $c_0 + c_1x$
- Klassen Parabola har også en table-metode - den er arvet
- `__init__` og `__call__` blir *overstyrt* eller *redefinert* i

Vi kan sjekke klassetype og klasserelasjoner med
`isinstance(obj, type)` og
`issubclass(subclassname, superclassname)`

```
>>> from Line_Parabola import Line, Parabola
>>> l = Line(-1, 1)
>>> isinstance(l, Line)
>>> True
>>> isinstance(l, Parabola)
>>> False

>>> p = Parabola(-1, 0, 10)
>>> isinstance(p, Parabola)
>>> True
>>> isinstance(p, Line)
>>> True

>>> issubclass(Parabola, Line)
>>> True
>>> issubclass(Line, Parabola)
>>> False

>>> p.__class__ == Parabola
>>> True
>>> p.__class__.__name__      # strengversjon av klassenavnet
>>> 'Parabola'
```

Line som en subklasse av Parabola

- Subklasser er ofte spesialtilfeller av en superklasse
- En linje $c_0 + c_1x$ er et spesialtilfelle av en parabel $c_0 + c_1x + c_2x^2$
- Kan Line være en subklasse av Parabola?
- Ikke noe problem - dette er opp til programmereren
- Mange vil si at dette er en bedre relasjon mellom disse to klassene

Kode for Line som subklasse av Parabola

```
class Parabola:
    def __init__(self, c0, c1, c2):
        self.c0, self.c1, self.c2 = c0, c1, c2

    def __call__(self, x):
        return self.c2*x**2 + self.c1*x + self.c0

    def table(self, L, R, n):
        """Returner en tabell med n punkter for L <= x <= R."""
        s = ''
        for x in np.linspace(L, R, n):
            y = self(x)
            s += f'{x:12g} {y:12g}\n'
        return s

class Line(Parabola):
    def __init__(self, c0, c1):
        super().__init__(self, c0, c1, 0)
```

Merk: `__call__` og `table` kan gjenbrukes i klassen `Line`!

Eksempel 2; klassen for numerisk derivasjon fra forrige uke

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

```
class Derivative:
    def __init__(self, f, h=1E-5):
        self.f = f
        self.h = float(h)

    def __call__(self, x):
        f, h = self.f, self.h    # lag korte former
        return (f(x+h) - f(x))/h

    def f(x):
        return exp(-x)*cos(tanh(x))

from math import exp, cos, tanh
dfdxdx = Derivative(f)
print dfdxdx(2.0)
```

Det finnes mange formler for numerisk derivasjon

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h)$$

$$f'(x) = \frac{f(x) - f(x-h)}{h} + O(h)$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

$$f'(x) = \frac{4}{3} \frac{f(x+h) - f(x-h)}{2h} - \frac{1}{3} \frac{f(x+2h) - f(x-2h)}{4h} + O(h^4)$$

$$f'(x) = \frac{3}{2} \frac{f(x+h) - f(x-h)}{2h} - \frac{3}{5} \frac{f(x+2h) - f(x-2h)}{4h} +$$

$$\frac{1}{10} \frac{f(x+3h) - f(x-3h)}{6h} + O(h^6)$$

$$f'(x) = \frac{1}{h} \left(-\frac{1}{6} f(x+2h) + f(x+h) - \frac{1}{2} f(x) - \frac{1}{3} f(x-h) \right) + O(h^3)$$

Hvordan kan vi lage en modul som tilbyr alle disse formlene?

Det er enkelt:

```
class Forward1:
    def __init__(self, f, h=1E-5):
        self.f = f
        self.h = float(h)

    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h

class Backward1:
    def __init__(self, f, h=1E-5):
        self.f = f
        self.h = float(h)

    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x) - f(x-h))/h

class Central2:
    # samme konstruktør
    # putt relevant formel i __call__
```

Hva er problemet med denne typen kode?

Alle konstruktørene er identiske, så vi dupliserer mye kode.

- Objekt-orientert tankegang: plasser kode som er felles for mange klasser i en superklasse, arv denne koden i mer spesialiserte klasser
- Her: arv konstruktøren fra superklasse, <linebreak> la subklasser for ulike derivasjonsformler implementere sin versjon av `__call__`

Klassehierarki for numerisk derivasjon

Superklasse:

```
class Diff:
    def __init__(self, f, h=1E-5):
        self.f = f
        self.h = float(h)
```

Subklasse for enkel 1. ordens foroverformel:

```
class Forward1(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        return (f(x+h) - f(x))/h
```

Subklasse for 4. ordens sentralformel:

```
class Central4(Diff):
    def __call__(self, x):
        f, h = self.f, self.h
        term1 = (4./3)*(f(x+h) - f(x-h)) / (2*h)
        term2 = (1./3)*(f(x+2*h) - f(x-2*h))/(4*h)
        return term1 - term2
```

Interaktivt eksempel: $f(x) = \sin x$, beregn $f'(x)$ for $x = \pi$

```
>>> from Diff import *  
>>> from math import sin  
>>> mycos = Central4(sin)  
>>> mycos(pi)  
>>> -1.000000082740371
```

`Central4(sin)` kaller den arvede konstruktøren i superklassen, mens `mycos(pi)` kaller `__call__` i subklassen `Central4`

Oppsummering av OOP-prinsipper

- En subklasse arver alt fra superklassen
- Når skal man bruke en subklasse/superklasse?
 - hvis kode som er felles for flere klasser kan plasseres i en superklasse
 - hvis problemet har naturlig hierarkisk struktur
- Programflyten hopper mellom super- og subklasser
- Typisk oppgave i OOP; når man lager en subklasse, undersøk superklassen for å identifisere delene som kan gjenbrukes, og hva som må legges til. Ofte kan subklassedefinisjonen være ganske kort!