

Klasser del 2 - spesialmetoder (kap 7)

Joakim Sundnes^{1,2}

Simula Research Laboratory¹

Universitetet i Oslo, Institutt for informatikk²

Oct 14, 2026

Torsdag 15/10:

- Oppgave 7.1, 7.2
- Mer om klasser:
 - Beskyttede attributter (innkapsling)
 - Spesialmetoder
- Eksempler på bruk av klasser og spesialmetoder:
 - Vektorer i planet
 - "Automatisk" derivasjon
 - (Klasse for polynomer - valgfritt selvstudium)
- Oppgave 7.10, 7.11

- Klassekonstruktøren har et spesielt navn: `__init__`
- Navnet gjenkjennes av Python, og sikrer at denne metoden kalles når en ny klasseinstans opprettes; f.eks. `y = MinKlasse(4)`
- Konstruktøren er et eksempel på en *spesialmetode*
- Spesialmetoder har navn med innledende og avsluttende doble understreker
- Spesialmetoder gjenkjennes av Python, og kalles automatisk når vi utfører ulike operasjoner på klasseinstansene

Husk klassen for å representere en funksjon:

```
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
```

```
y = Y(3)
v = y.value(0.1)
```

Men det ville vært mer naturlig å bruke klassen slik:

```
y = Y(3)
v = y(0.1)
```

Spesialmetoden call; implementasjon

Gi value-metoden nytt navn: `__call__`

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

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

Nå kan vi skrive

```
y = Y(3)
v = y(0.1) # samme som v = y.__call__(0.1) eller Y.__call__(y, 0.1)
```

Merk:

- Instansen `y` oppfører seg nå som en vanlig funksjon!
- Instansen har blitt *kallbar* (*callable*)
- `value(t)`-metoden gjør akkurat det samme, men `__call__` gir penere syntaks.

Spesialmetode for utskrift

- Vi er vant til å skrive ut et objekt `a` med `print(a)`, som fungerer for innebygde typer (strenger, lister, flyttall, ...)
- For brukerdefinerte klasser vet ikke Python hva som skal skrives ut. Vi må spesifisere det selv ved å legge til en spesialmetode `__str__` i klassen

Eksempel:

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

    def __str__(self):
        return f'v0*t - 0.5*g*t**2; v0={self.v0}'
```

Bruk:

```
>>> y = Y(1.5)
>>> y(0.2)
>>> 0.1038
>>> print(y)
>>> v0*t - 0.5*g*t**2; v0=1.5
```

Klasse Y på nytt med utskriftsmetode

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

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

    def __str__(self):
        return f'v0*t - 0.5*g*t**2; v0={self.v0}'
```

Spesialmetoder for aritmetiske operasjoner

```
c = a + b    # c = a.__add__(b)
c = a - b    # c = a.__sub__(b)
c = a * b     # c = a.__mul__(b)
c = a / b     # c = a.__div__(b)
c = a**e     # c = a.__pow__(e)
```

Spesialmetoder for sammenligning

`a == b` `# a.__eq__(b)`

`a != b` `# a.__ne__(b)`

`a < b` `# a.__lt__(b)`

`a <= b` `# a.__le__(b)`

`a > b` `# a.__gt__(b)`

`a >= b` `# a.__ge__(b)`

Programmereren har ansvaret for å definere spesialmetoder!

Hvordan bør for eksempel `__add__(self, other)` og `__mul__(self, other)` defineres?

- Dette er helt opp til den som skriver funksjonene
- Bør implementere noe som oppfattes som meningsfulle resultater av `objekt1 + objekt2` og `objekt1 * objekt2`.

Eksempel; klasse for vektorer i planet

Matematiske operasjoner for vektorer i planet:

$$(a, b) + (c, d) = (a + c, b + d)$$

$$(a, b) - (c, d) = (a - c, b - d)$$

$$(a, b) \cdot (c, d) = ac + bd$$

$$(a, b) = (c, d) \text{ hvis } a = c \text{ og } b = d$$

Ønsket bruk:

```
>>> u = Vec2D(0,1)
>>> v = Vec2D(1,0)
>>> print(u + v)
>>> (1, 1)
>>> a = u + v
>>> w = Vec2D(1,1)
>>> a == w
>>> True
>>> print(u - v)
>>> (-1, 1)
>>> print(u*v)
>>> 0
```

Klasse for vektorer; implementasjon

```
class Vec2D:
    def __init__(self, x, y):
        self.x = x; self.y = y

    def __add__(self, other):
        return Vec2D(self.x+other.x, self.y+other.y)

    def __sub__(self, other):
        return Vec2D(self.x-other.x, self.y-other.y)

    def __mul__(self, other):
        return self.x*other.x + self.y*other.y

    def __abs__(self):
        return math.sqrt(self.x**2 + self.y**2)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __str__(self):
        return f'({self.x}, {self.y})'
```

Eksempel; "automatisk" derivasjon

Gitt en matematisk funksjon i Python, for eksempel

```
def f(x):  
    return x**3
```

kan vi lage en klasse Derivative og skrive

```
dfdxdx = Derivative(f)
```

slik at dfdx oppfører seg som en funksjon som beregner den deriverte av $f(x)$?

```
print(dfdx(2))    # beregner  $3*x**2$  for  $x=2$ 
```

Metode

Vi estimerer den deriverte numerisk:

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

for en liten h , for eksempel $h = 10^{-5}$

Implementasjon

```
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
```

Metode

Vi estimerer den deriverte numerisk:

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

for en liten h , for eksempel $h = 10^{-5}$

Implementasjon

```
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
```

Metode

Vi estimerer den deriverte numerisk:

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

for en liten h , for eksempel $h = 10^{-5}$

Implementasjon

```
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
```

Automatisk derivasjon; bruk

```
>>> from math import *
>>> df = Derivative(sin)
>>> x = pi
>>> df(x)
>>> -1.000000082740371
>>> cos(x) # eksakt
>>> -1.0
>>> def g(t):
>>> ...     return t**3
>>> ...
>>> dg = Derivative(g)
>>> t = 1
>>> dg(t) # sammenlign med 3 (eksakt)
>>> 3.000000248221113
```

Automatisk derivasjon; nyttig i Newtons metode

Newtons metode løser ligninger på formen $f(x) = 0$, men metoden krever $f'(x)$

```
def Newton(f, dfdx, x0, epsilon=1E-6):  
    ...  
    return x, konvertert, iter
```

Anta at $f'(x)$ er komplisert eller tidkrevende å regne ut. Da er klassen Derivative hendig:

```
>>> def f(x):  
>>> ...     return 100000*(x - 0.9)**2 * (x - 1.1)**3  
>>> ...  
>>> df = Derivative(f)  
>>> xstart = 1.01  
>>> Newton(f, df, xstart, epsilon=1E-5)  
>>> (1.0987610068093443, True, 8)
```

Klasse for polynomer; funksjonalitet

Et polynom kan spesifiseres med en liste av koeffisientene sine. For eksempel, $1 - x^2 + 2x^3$ er

$$1 + 0 \cdot x - 1 \cdot x^2 + 2 \cdot x^3$$

og koeffisientene kan lagres som $[1, 0, -1, 2]$

Ønsket applikasjonskode:

```
>>> p1 = Polynomial([1, -1])
>>> print(p1)
>>> 1 - x
>>> print(p1(x=0.5))
>>> 0.5
>>> p2 = Polynomial([0, 1, 0, 0, -6, -1])
>>> p3 = (p1 + p2)
>>> print(p3.coeff)
>>> [1, 0, 0, 0, -6, -1]
>>> print(p3)
>>> 1 - 6*x^4 - x^5
>>> p2.differentiate()
>>> print(p2)
>>> 1 - 24*x^3 - 5*x^4
```

Hvordan kan vi lage klassen Polynomial?

Polynomial-klassen; første del

```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s
```

Polynomial-klassen; addisjon

```
class Polynomial:
    ...

    def __add__(self, other):
        # returner self + other

        # start med den lengste listen og legg til den andre:
        if len(self.coeff) > len(other.coeff):
            coeffsum = self.coeff[:] # kopier!
            for i in range(len(other.coeff)):
                coeffsum[i] += other.coeff[i]
        else:
            coeffsum = other.coeff[:] # kopier!
            for i in range(len(self.coeff)):
                coeffsum[i] += self.coeff[i]
        return Polynomial(coeffsum)
```

Polynomial-klassen; multiplikasjon

Matematikk:

Multiplikasjon av to generelle polynomer:

$$\left(\sum_{i=0}^M c_i x^i \right) \left(\sum_{j=0}^N d_j x^j \right) = \sum_{i=0}^M \sum_{j=0}^N c_i d_j x^{i+j}$$

Koeffisienten som korresponderer til potensen $i + j$ er $c_i \cdot d_j$. Listen r med koeffisientene til resultatet: $r[i+j] = c[i]*d[j]$ (i og j løper fra 0 til M og N , henholdsvis)

Implementasjon:

```
class Polynomial:
    ...
    def __mul__(self, other):
        M = len(self.coeff) - 1
        N = len(other.coeff) - 1
        coeff = [0]*(M+N+1) # eller np.zeros(M+N+1)
        for i in range(0, M+1):
            for j in range(0, N+1):
                coeff[i+j] += self.coeff[i]*other.coeff[j]
        return Polynomial(coeff)
```

Polynomial-klassen; multiplikasjon

Matematikk:

Multiplikasjon av to generelle polynomer:

$$\left(\sum_{i=0}^M c_i x^i \right) \left(\sum_{j=0}^N d_j x^j \right) = \sum_{i=0}^M \sum_{j=0}^N c_i d_j x^{i+j}$$

Koeffisienten som korresponderer til potensen $i + j$ er $c_i \cdot d_j$. Listen `r` med koeffisientene til resultatet: `r[i+j] = c[i]*d[j]` (i og j løper fra 0 til M og N , henholdsvis)

Implementasjon:

```
class Polynomial:
    ...
    def __mul__(self, other):
        M = len(self.coeff) - 1
        N = len(other.coeff) - 1
        coeff = [0]*(M+N+1) # eller np.zeros(M+N+1)
        for i in range(0, M+1):
            for j in range(0, N+1):
                coeff[i+j] += self.coeff[i]*other.coeff[j]
        return Polynomial(coeff)
```

Polynomial-klassen; multiplikasjon

Matematikk:

Multiplikasjon av to generelle polynomer:

$$\left(\sum_{i=0}^M c_i x^i \right) \left(\sum_{j=0}^N d_j x^j \right) = \sum_{i=0}^M \sum_{j=0}^N c_i d_j x^{i+j}$$

Koeffisienten som korresponderer til potensen $i + j$ er $c_i \cdot d_j$. Listen r med koeffisientene til resultatet: $r[i+j] = c[i]*d[j]$ (i og j løper fra 0 til M og N , henholdsvis)

Implementasjon:

```
class Polynomial:
    ...
    def __mul__(self, other):
        M = len(self.coeff) - 1
        N = len(other.coeff) - 1
        coeff = [0]*(M+N+1) # eller np.zeros(M+N+1)
        for i in range(0, M+1):
            for j in range(0, N+1):
                coeff[i+j] += self.coeff[i]*other.coeff[j]
        return Polynomial(coeff)
```

Klasse Polynomial; derivasjon

Matematikk:

Regel for å derivere et generelt polynom:

$$\frac{d}{dx} \sum_{i=0}^n c_i x^i = \sum_{i=1}^n i c_i x^{i-1}$$

Hvis c er listen med koeffisienter, har den deriverte en liste med koeffisienter, dc , der $dc[i-1] = i*c[i]$ for i som løper fra 1 til den største indeksen i c . Merk at dc har ett element mindre enn c .

Implementasjon:

```
class Polynomial:
    ...
    def differentiate(self):      # endre self
        for i in range(1, len(self.coeff)):
            self.coeff[i-1] = i*self.coeff[i]
        del self.coeff[-1]

    def derivative(self):        # returner nytt polynom
        dpdx = Polynomial(self.coeff[:]) # kopier
        dpdx.differentiate()
        return dpdx
```

Polynomial-klassen; pen utskrift

```
class Polynomial:
    ...
    def __str__(self):
        s = ''
        for i in range(0, len(self.coeff)):
            if self.coeff[i] != 0:
                s += f' + {self.coeff[i]}*x^{i}'

        # fiks layout (mange spesialtilfeller):
        s = s.replace('+ -', '- ')
        s = s.replace(' 1*', ' ')
        s = s.replace('x^0', '1')
        s = s.replace('x^1 ', 'x ')
        s = s.replace('x^1', 'x')
        if s[0:3] == ' + ': # fjern innledende +
            s = s[3:]
        if s[0:3] == ' - ': # fiks mellomrom for innledende -
            s = '- ' + s[3:]
        return s
```

Polynomial-klassen; eksempel på bruk

Vi har to polynomer:

$$p_1(x) = 1 - x, \quad p_2(x) = x - 6x^4 - x^5.$$

Summen av disse er

$$p_3(x) = p_1(x) + p_2(x) = 1 - 6x^4 - x^5$$

```
>>> p1 = Polynomial([1, -1])
>>> print(p1)
>>> 1 - x
>>> p2 = Polynomial([0, 1, 0, 0, -6, -1])
>>> p3 = (p1 + p2)
>>> print p3.coeff
>>> [1, 0, 0, 0, -6, -1]
>>> p2.differentiate()
>>> print(p2)
>>> 1 - 24*x^3 - 5*x^4
```

- Klasser pakker sammen data og funksjoner som naturlig hører sammen
- Vi definerer en klasse og lager deretter *instanser* (eller objekter) av klassen
 - Ulike instanser vil ha ulike data, men de har alle de samme metodene (funksjonene) som opererer på dataene
- I IN1900-programmer er klasser aldri strengt nødvendig, men i noen tilfeller er de veldig praktiske å bruke
- I «virkelige» programmer, med titusenvise linjer, kan den ekstra organiseringen som klasser tilbyr være forskjellen på en kode som virker og en som ikke gjør det

Spesialmetoder - oppsummering

Spesialmetoder er metoder med spesielle navn, som gjør at de automatisk kalles når vi utfører ulike operasjoner på instanser av klassen. Tre av de viktigste er:

- `__init__`: Kalles automatisk hver gang vi lager en ny instans av klassen (kalles ofte en konstruktør)
- `__call__`: Gjør instanser av klassen *kallbare*, dvs at vi kan bruke dem som vanlige Python-funksjoner
- `__str__`: Gir pen utskrift av instanser (denne metoden bestemmer hva som skal skrives ut om en instans)

I tillegg har vi spesialmetoder for aritmetiske operasjoner, lengde, indeksering, etc:

- `c = a + b` kaller på `c = a.__add__(b)`
- Ulike spesialmetoder kalles automatisk ved operasjoner som `a+b`, `a-b`, `a*b`, `a/b`, `a**b`, `-a`, `if a:`, `len(a)`, etc