Ch.9: Objektorientert programmering

Ole Christian Lingjærde, Institutt for Informatikk, UiO

23. oktober 2018

Dagens agenda

- Mer om klasser
- Objektorientert programmering

Er du der du skal være?

Du bør nå skjønne ganske nøyaktig hva denne klassen gjør:

```
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(x) = %g * sin(%g*x)' % (self.v0, self.v1)

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

Klarer du å svare på følgende spørsmål?

- Hvor mange attributter og hvor mange metoder har klassen?
- Hvorfor har noen metoder navn av formen `__XXXX__`?
- Hva er variabelen `self`?
- Hvilke metoder i klassen utføres i hver linje nedenfor?

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

Svar

- To attributter (`v0` og `v1`) og fire metoder
- Metoder med navn `__XXXX__` kalles *spesialmetoder*. Kall på slike metoder har spesiell syntaks (mer om det senere i dag)
- Variabelen `self` er en intern peker til objektet/instansen. Dette er samme peker som returneres til "utsiden" når objektet opprettes.
- Metodekall som utføres:

```
f = Fnc(7, 2)       # Kaller på: __init__
print(f)            # Kaller på: __str__
print(f(3.5))       # Kaller på: __call__
print(f.df(3.5))    # Kaller på: df
```

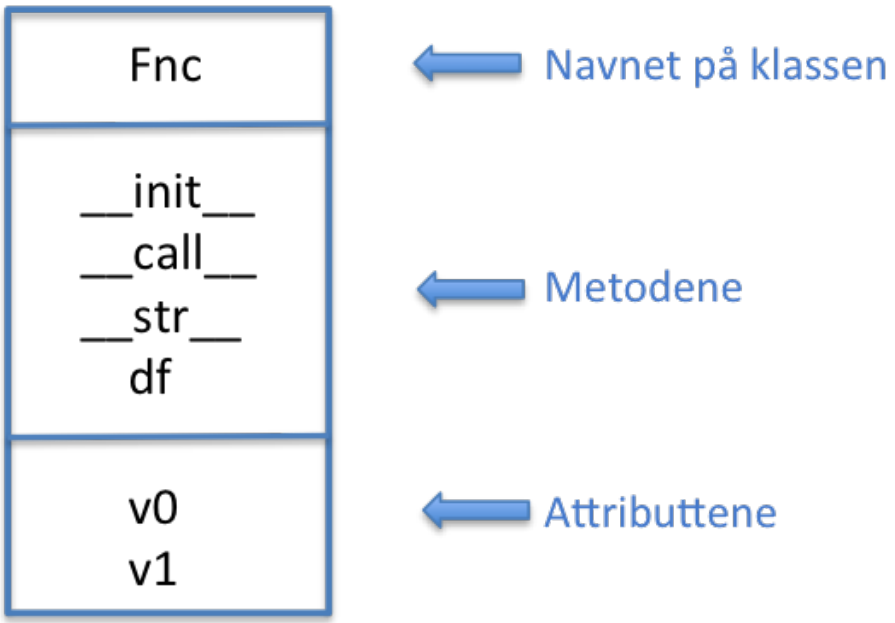
- Klarte du alle spørsmålene? Da er du klar for å gå videre!
- Møtte du problemer? Ennå ikke for sent, men *veldig viktig* at du nå tar igjen etterslepet.
- I dag: bygge videre på det vi har lært om klasser til nå.

UML-diagrammer

UML (Unified Modeling Language) er en visuell måte å fremstille og dokumentere datamodeller på.

Vi kan bruke UML-diagrammer til å visualisere innholdet i klasser, og relasjoner mellom klasser.

Eksempel (for klassen vi nettopp definerte):



Spesialmetoder

- Spesialmetoder i klasser er strengt tatt aldri nødvendige å bruke
- Men de gjør ofte at programmer blir mer elegante
- Krever litt trening å bli kjent med syntaksen

Eksempel A

- Vi ønsker å definere en klasse `Complex` hvor objekter representerer komplekse tall med realdel og imaginærdel.
- De komplekse tallene $x = 1 + 2i$ og $z = 2 + i$ skal kunne lages slik: `x = Complex(1,2)` og `z = Complex(2,1)`.
- Vi ønsker å kunne summere to komplekse tall. Vi kunne laget en klassemetode `add(self,z)` slik at `x.add(z)` beregner summen $x + z$.
- Bedre alternativ: kall metoden `__add__(self,z)`. Da kan vi addere i Python ved å skrive `x + z`.

Eksempel A: Matematiske formler

La $x = a + bi$ og $z = c + di$ være to komplekse tall. Da er:

$$\begin{aligned} x + z &= (a + c) + (b + d)i \\ x - z &= (a - c) + (b - d)i \\ x * z &= (ac - bd) + (ad + bc)i \\ x/z &= (ac + bd) + (bc - ad)i \end{aligned}$$

Hvis klassen `Complex` har attributter `real` og `imag` kan vi dermed få utført addisjonen `__add__(self,z)` slik:

```
real = self.real + z.real
imag = self.imag + z.imag
```

Eksempel A: Python-kode

```
class Complex:
    def __init__(self, real, imag):
        self.real = float(real)
        self.imag = float(imag)

    def __str__(self):
        s = '%g + %gi' % (self.real, self.imag)
        return s

    def __add__(self, z):
        real = self.real + z.real
        imag = self.imag + z.imag
        res = Complex(real, imag)
        return res

# Eksempel på bruk:
x = Complex(1,2)
y = Complex(2,1)
z = x + y   # Ekvivalent: z = Complex.__add__(x,y)
print(z)
```

Eksempel B

- Anta at vi har klassen `Complex` fra forrige eksempel
- Vi kan bruke samme "trikset" til å definere alle fire regnearter
- Vi kan bruke spesialfunksjonene `__sub__`, `__mul__` og `__div__` på samme måte som vi har brukt `__add__`

Eksempel B: Python-kode

```
class Complex:
    <Alt tidligere som før>

    def __sub__(self, z):
        real = self.real - z.real
        imag = self.imag - z.imag
        res = Complex(real, imag)
        return res

    def __mul__(self, z):
        real = self.real*z.real - self.imag*z.imag
        imag = self.real*z.imag + self.imag*z.real
        res = Complex(real, imag)
        return res

    def __div__(self, z):
        r = z.real**2 + z.imag**2
        real = (self.real*z.real + self.imag*z.imag)/r
        imag = (self.imag*z.real - self.real*z.imag)/r
        res = Complex(real, imag)
        return res
```

Eksempler på bruk

```
x = Complex(1,1)    # x = 1 + i
y = Complex(2,3)    # y = 2 + 3i

print(x + y)        # 3 + 4i
print(x - y)        # -1 - 2i
print(x * y)        # -1 + 5i
print(x / y)        # 0.384615 + -0.0769231i

print(x * y / y)    # 1 + 1i
```

Noen viktige spesialmetoder

```
a.__init__(self, args)        # Konstruktør
a.__del__(self)               # Destruktør
a.__call__(self, args)        # Funksjonskall
a.__str__(self)               # Tekstrepresentasjon
a.__repr__(self)              # a = eval(repr(a))
a.__add__(self, b)            # a + b
a.__sub__(self, b)            # a - b
a.__mul__(self, b)            # a * b
a.__div__(self, b)            # a / b
a.__pow__(self, b)            # a ** b
a.__lt__(self, b)             # a < b
a.__le__(self, b)             # a <= b
a.__gt__(self, b)             # a > b
a.__ge__(self, b)             # a >= b
a.__eq__(self, b)             # a == b
a.__ne__(self, b)             # a != b
```

Objektorientert programmering (OOP)

- Alt i Python er objekter, så teknisk sett er all Python-programmering objektbasert.
- I objektorientert programmering (OOP) går vi ett skritt videre.
- OOP utnytter en svært nyttig egenskap ved klasser: de kan settes sammen som byggeklosser!
- Hvis vi har definert en klasse `class A` så kan vi definere en ny klasse `class B(A)`.
- Da blir klassen `B` en *utvidelse* av klassen `A`
- Vi sier at `B` *arver* data og metoder fra `A`
- Vi sier også at `B` er subklasse av `A`, og at `A` er superklasse til `B`

Prinsipp A: Klasser kan arve fra andre klasser

```
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
```

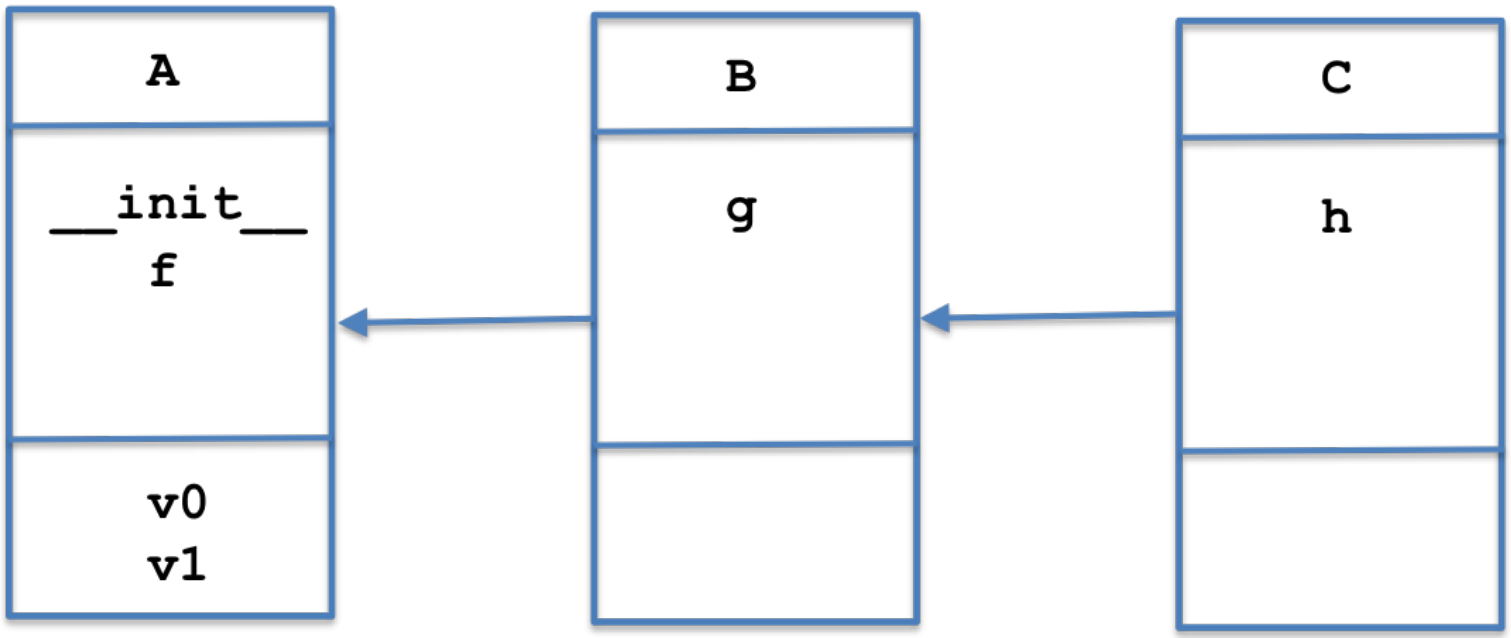
Vi har nå definert tre klasser:

`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`)

UML-diagram



Innholdet i klassene A, B og C

```
# I objekter av A har vi attributtene v0, v1 og metoden f:
p = A(2.7, 5.2)
print(p.v0)         # Utskrift: 2.7
print(p.v1)         # Utskrift: 5.2
print(p.f(3.0))     # Utskrift: 9.0

# I objekter av B har vi det samme + metoden g:
p = B(2.7, 5.2)
print(p.v0)         # Utskrift: 2.7
print(p.v1)         # Utskrift: 5.2
print(p.f(3.0))     # Utskrift: 9.0
print(p.g(3.0))     # Utskrift: 81.0

# I objekter av C har vi det samme + metoden h:
p = C(2.7, 5.2)
print(p.v0)         # Utskrift: 2.7
print(p.v1)         # Utskrift: 5.2
print(p.f(3.0))     # Utskrift: 9.0
print(p.g(3.0))     # Utskrift: 81.0
print(p.h(3.0))     # Utskrift: 729.0
```

Prinsipp B: Subklasser kan overkjøre metoder i superklasser

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

    def skrivut(self):
        print("Klasse A")

class B(A):
    def __init__(self, b):    # Overkjører __init__ i class A
        self.b = b

    def skrivut(self):        # Overkjører skrivut i class A
        print("Klasse B")

# Eksempler på bruk
p = A(3)      # Lag objekt av superklassen A
p.skrivut()   # Utskrift: "Klasse A"
p = B(4)      # Lag objekt av subklassen B
p.skrivut()   # Utskrift: "Klasse B"
```

Hensikten med å overkjøre metoder

- Subklasser kan brukes for å legge til ny funksjonalitet. Da kan det tenkes at vi har behov for å definere enda flere instansvariable med konstruktøren. Da må denne defineres på nytt i subklassen.
- Subklasser kan også brukes for å restriktere funksjonaliteten i klassen det arves fra. Da kan det tenkes at vi trenger å initialisere noen av instansvariablene i superklassen til 0. Da må konstruktøren også defineres på nytt.
- Utskrift og andre funksjoner kan også være nødvendig å endre i subklasser.
- Kommer til å bli klarere etterhvert som vi ser eksempler

Prinsipp C: Overkjørte metoder finnes fortsatt

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

    def skrivut(self):
        print("Klasse A")

class B(A):
    def __init__(self, a, b):
        A.__init__(self, a)    # Kall __init__ i superklassen
        self.b = b

    def skrivut(self):
        A.skrivut(self)     # Kall skrivut i superklassen
        print("Klasse B")

p = B(3,4)  # Lag objekt av subklassen B
p.skrivut() # Utskrift: 'Klasse A' + linjeskift + 'Klasse B'
```

Prinsipp D: Vi kan ha mange nivåer av subklasser

```
class A:
    def __init__(self, a):
        self.a = a
    def skrivut(self):
        print('a = %g' % self.a)

class B(A):
    def __init__(self, a, b):
        A.__init__(self, a)
        self.b = b
    def skrivut(self):
        print('a = %g, b = %g' % (self.a, self.b))

class C(B):
    def __init__(self, a, b, c):
        B.__init__(self, a, b)
        self.c = c
    def skrivut(self):
        print('a = %g, b = %g, c = %g' % (self.a, self.b, self.c))

p1 = A(1)
p1.skrivut()      # a = 1
p2 = B(1,2)
p2.skrivut()      # a = 1, b = 2
p3 = C(1,2,3)
p3.skrivut()      # a = 1, b = 2, c = 3
```

Prinsipp E: Vi kan alltid finne ut hvor vi er i hierarkiet

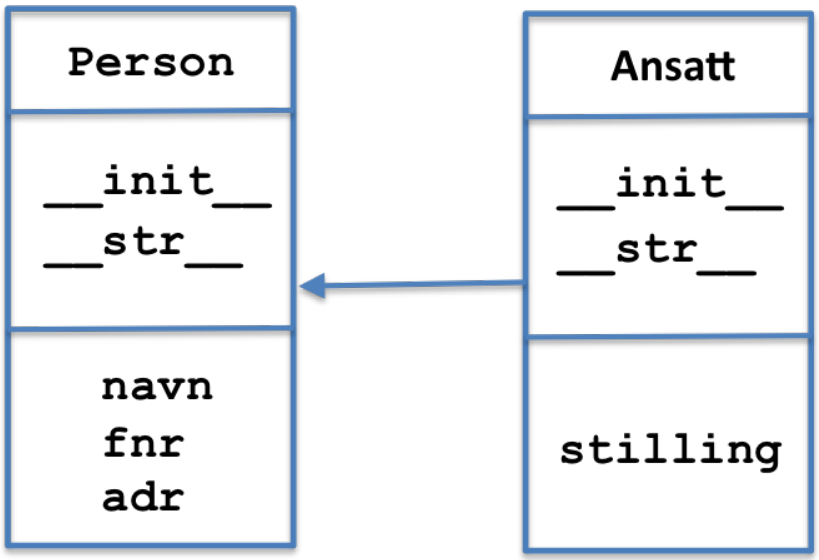
Anta at klassene A, B, C er definert som på forrige slide.

```
# Lag objekter av A og B
p = A(1)
q = B(1,2)

# Hvilke klasser er et objekt en instans av?
isinstance(p, A)  # True
isinstance(p, B)  # False
isinstance(q, A)  # True
isinstance(q, B)  # True

# Hvilken unike klasse tilhører objektet p?
print(p.__class__ == A)     # True
print(p.__class__ == B)     # False
print(q.__class__ == A)     # False
print(q.__class__ == B)     # True

# Alternativ til over
print(p.__class__.__name__ == 'A')    #  True

# Er klassen B en subklasse av klassen A?
issubclass(B, A)  # True
issubclass(A, B)  # False

# Finn navnet på superklassen til et objekt
print(q.__class__.__bases__[0].__name__)  # A
```

Eksempel A: Person <- Ansatt

```
class Person:
    def __init__(self, navn, fnr, adr):
        self.navn = navn
        self.fnr = fnr
        self.adr = adr

    def __str__(self):
        s = 'Navn: %s\nFnr: %s\nAdresse: %s\n' % \
        (self.navn, self.fnr, self.adr)
        return s

class Ansatt(Person):
    def __init__(self, navn, fnr, adr, stilling):
        Person.__init__(self, navn, fnr, adr)
        self.stilling = stilling

    def __str__(self):
        s1 = Person.__str__(self)
        s2 = 'Stilling: %s\n' % self.stilling
        return(s1 + s2)

p = Person('Rex', '18050012345', 'Slottet')
print(p)

p = Ansatt('Rex', '18050012345', 'Slottet', 'Konge')
print(p)
```

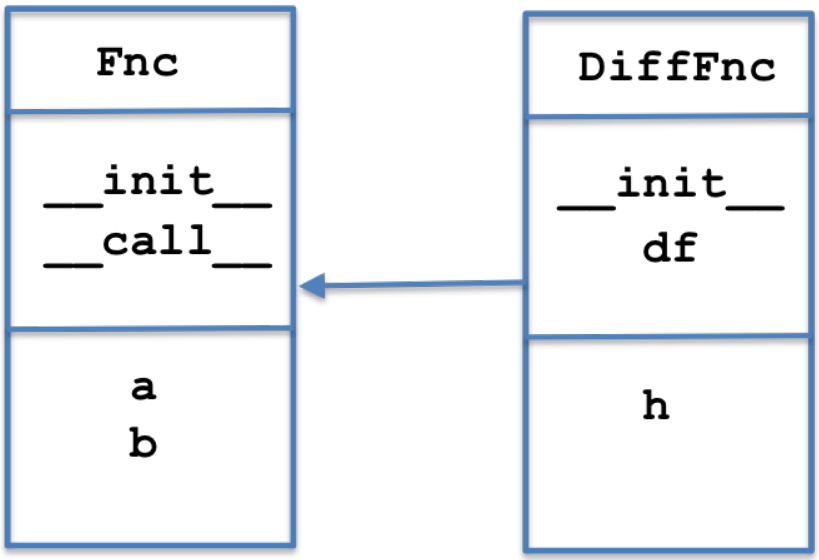
UML-diagram



Eksempel B: Fnc <- DiffFnc

```
from math import sin, cos

class Fnc:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __call__(self, x):
        res = self.a*sin(x) + self.b*cos(x)
        return res

class DiffFnc(Fnc):
    def __init__(self, a, b, h=1E-5):
        Fnc.__init__(self, a, b)
        self.h = h

    def df(self, x):
        res = self.a*cos(x) - self.b*sin(x)
        return res

#
f = Fnc(2,6)
print(f(3))          # -5.65771496348
#
f = DiffFnc(2,6)
print(f(3))          # -5.65771496348
print(f.df(3))       # -2.82670504156
```

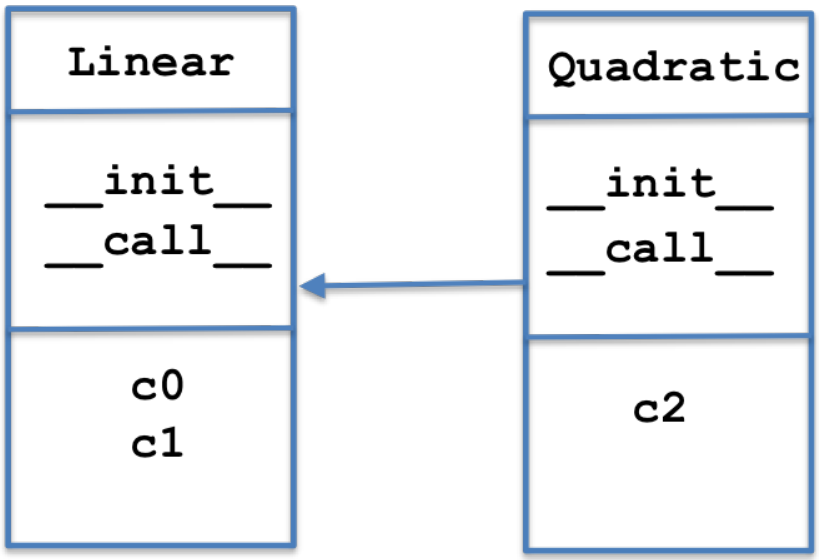
UML-diagram



Eksempel C: Linear <- Quadratic

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

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

class Quadratic(Linear):
    def __init__(self, c0, c1, c2):
        Linear.__init__(self, c0, c1)
        self.c2 = c2

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

# Test av klassene
p = Quadratic(3, 4, 5)
print(p(2.5))              # 44.25
```

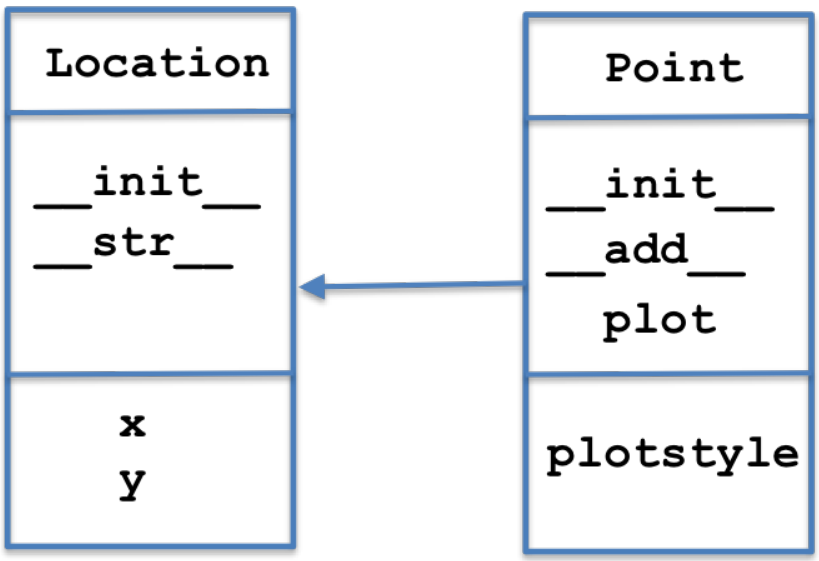
UML-diagram



Eksempel D: Location <- Point

```
import matplotlib.pyplot as plt

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

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)

class Point(Location):
    def __init__(self, x, y, plotstyle='ro'):
        Location.__init__(self, x, y)
        self.plotstyle = plotstyle

    def __add__(self, p):
        xnew = self.x + p.x
        ynew = self.y + p.y
        return Point(xnew, ynew)

    def plot(self):
        plt.plot(self.x, self.y, self.plotstyle)

p1 = Point(3,4); p2 = Point(1,1); p3 = Point(2.5, 1.5)
p4 = p1 + p3
p1.plot(); p2.plot(); p3.plot(); p4.plot()
```

UML-diagram



Eksempel E: Derivatives <- SinCos

```
from math import sin, cos

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

    def __call__(self, x):
        print('Not implemented in this class!')

    def df(self, x):
        return (self(x+self.h)-self(x-self.h))/(2*self.h)

    def ddf(self, x):
        return (self(x+self.h)-2*self(x)+self(x-self.h))/(self.h**2)

class SinCos(Derivatives):
    def __init__(self, a, b):
        Derivatives.__init__(self)
        self.a = a; self.b = b

    def __call__(self, x):
        return self.a * cos(x) + self.b * sin(x)

f = SinCos(3, 6); x = 0.5
print('f(%g) = %g' % (x, f(x)))
print('df(%g) = %g' % (x, f.df(x)))
print('ddf(%g) = %g' % (x, f.ddf(x)))
```

UML-diagram