FY1008_M04

September 23, 2025

1 Module 04: Funksjoner og Testing av Kode

Denne modulen svarer til kapittelet 4 i læreboken.

Funksjoner er struktur som de fleste programmeringsspråk har. De er fine å bruke for å strukturere koden din. Dessuten er de fine å bruke når man skal feilsøke (debugge) koden da de ofte tvinger programmereren til å dele opp problemet i mindre deler. Disse delene kan man da teste isolert fra resten av programmet.

Vi har tidligere sette hvordan man definerer funksjoner, men her skal vi gå mer systematisk gjennom dem.

En funksjon i python har følgende struktur

def (argumenter): “ “ ” Dokumentasjon (docstring) “ “ ”
return

En funksjon kan returnere mer eller mindre hva som helst. Det kan være en skalar, en listen, en numpy array, et egendefinert klasse objekt etc, eller en vilkårlig kombinasjon av slike.

1.1 En standard definisjon

[1]:
```
# Et første eksempel
def square(x):
    x_square = x*x
    return x_square
# end function
```

[ ]:
```
x_int = 2
print("x_square er :", square(x_int) )

x_float = 3.
print("x_square er :", square(x_float) )

x_cmplx = 3j
print("x_square er :", square(x_cmplx) )
```

```
x_square er : 4
x_square er : 9.0
x_square er : (-9+0j)
```

Man kan kalle funksjonen med en vilkårlig type argument, så lenge som beregningen som gjøres i funksjonen er veldefinert for den typen man kaller den med.

For eksempel, kan vi bruke en ndarray som argument, som vist her:

[ ]:
```
import numpy as np

x_np = np.arange(4)

print("x_np        :", x_np )
print("x_square er :", square(x_np) )
```

En funksjon kan også ha flere argumenter:

[12]:
```
def a_times_b(a,b):
    res = a*b
    return res
# end function
```

[ ]:
```
print("x         :", x_np )
print("x_square  :", square(x_np) )
print("a_times_b :", a_times_b(x_np, x_np) )
```

Det er en god praksis lage en docstring for sine funksjoner. Her beskriver man hva funksjonen gjør sant hva input og output er. Gi gjerne også eksempler på hvordan man bruk av funksjonen. Det er denne informasjonen som vises når kan bruker help(). I code med autoDocstring installert er man godt igang ved å skrive " " " + enter (prøv dette)

[ ]:
```
def a_times_b(a,b):

    res = a*b
    return res
# end function
```

Det er to måter (ihvertfall) for å kalle en funksjon. Man kan bruke hva som kalles for *posisjonal arguments* eller *keyword arguments*.... Følgende eksempel viser dette....

[ ]:
```
def mydivide(a,b):
    res = a/b
    return res
# end function

aa = 2.
bb = 3.

print("Posisjonal argument   : ", mydivide(aa,bb) )
print("Posisjonal argument 2 : ", mydivide(bb,aa) )

print("Keyword argument    : ", mydivide(a=aa,b=bb) )
print("Keyword argument  2 : ", mydivide(b=bb,a=aa) )
```

```
print("Keyword argument  2 : ", mydivide(a=bb,b=aa) )
```

Studer følgende eksempel som beregner rente utviklingen etter n år for et beløp P når renten er r :

[ ]:
```
P = 100
r = 5.0

def amount(n):
    res = P * (1 + r/100 )**n
    return res
# end function

# etter 7 år
print(" Amount : ", amount(7) )
```

Here blir variablene P or rentesatsen r brukt inni funksjonen. Vi sier at de er i *the global namespace*. Hva skjer i dette tilfellet?

[ ]:
```
P = 100
r = 5.0

def amount(n):
    r = 4.0                     # <--- NY LINJE
    res = P * (1 + r/100 )**n
    return res
# end function

# etter 7 år
print(" Amount : ", amount(7) )
```

Vi ser vi får forskjellig resultat. Dette skyldes at r er tilgjengelig i *the local namespace* og og det lokale har høyere prioritet enn det globale.

[ ]:
```
P = 100
r = 5.0

def amount_2(n,r):              # <--- Endring
    res = P * (1 + r/100 )**n
    return res
# end function

# etter 7 år
print(" Amount : ", amount_2(7,4.) )
```

En funksjon kan også to *optional arguments*, dvs. argumenter som man ikke trenger spesifisere og om vi ikke gj;r det tar de en default verdi. Dette eksempelet viser dette....

[ ]:
```
P = 100
r = 5.0
```

```
def amount_2(n,r=4.):              # <--- Endring
    res = P * (1 + r/100 )**n
    return res
# end function

# etter 7 år
print(" Amount 1 : ", amount_2(7) )
print(" Amount 2 : ", amount_2(7,r) )
```

WARNING: Å bruke variable fra the globale namespace blir ofte ansett som, DÅRLIG programmeringspraksis. Årsaken er at funksjonen trenger mer data enn hva som kommer inn til funksjonen som argumenter og det er derfor ikke lett å holde oversikt over hvilke variable som trengs for at funksjonen skal fungere. For små enkle funksjoner er trolig dette ikke noe problem.

Om man virkelig vil, kan man oppdatere den globale verdien til r fra inne i funksjonen. Dette gjøres ved å definere *global r* samt å sette eb verdi for denne variabelen inne i funksjonen. Når dette er sagt, vil jeg legge til at dette er noe du bør unngå å gjøre selv om det formelt vil fungere!!!!

En bedre kode for å beregne rente er:

[ ]:
```
P = 100
r = 5.0

def amount(P,r,n):
    """Beregner veksten av en investering over tid

    Args:
        P (float): beløp
        r (float): rentesatsen
        n (float): tid i år
    """
    res = P * (1 + r/100 )**n
    return res
# end function

# Parameter verdier
P = 100
r = 5.0
n = 7

# Utskrift
print(" Amount : ", amount(P,r,n) )
```

```
 Amount :  140.71004226562505
```

I denne funksjonen kommer alt som trengs for å beregne resultatet som input til funksjonen. **Dette er en god praksis!!!!!**

1.2 Returnering av multiple argumenter, eller ingen argumenter

En funksjon kan også returnere flere argumenter, og følgende (tåpelige) eksempel illustrer dette:

[ ]:
```
def mysum(a,b,c):
    sum1 = a+b
    sum2 = b+c
    return sum1,sum2
# end function

s1, s2 = mysum(1,2,3)

# etter 7 år
print(" Sum1 : ", s1 )
print(" Sum2 : ", s2 )
```

Eller som du brukte når du skulle løse diff.likninger:

[ ]:
```
def yfunc(t,v0):
    g = 9.81
    y + v0*t -0.5*g*t**2
    dydt = v0 -g*t
    return y, dydt
# end function
```

[ ]:
```

```

En fusjon trenger strengt tatt ikke å returnere noe som helst. For eksempel, kan dette være aktuelt om man bare ønsker å få skrevet ut noe informasjon ala en hjelpe funksjon eller hvile parametre man bruker i beregningene.

[ ]:
```
def usage(version=None):
    print("Simuleringen ble kjørt med python koden MyCode! ")
    if version is not None:
        print("Versjon:",version)
    # end if
#end function

usage()
print("\n")

usage("1.2.4")
```

1.3 Funksjoner som argument til andre funksjoner

Det er ikke uvanlig at funksjoner brukes som argument til andre funksjoner. For eksempel, vil vi senere, se at dette er vanlig i mange scipy funksjoner.

Her vil vi istedet se på hvordan vi kan beregne deriverte av en funksjon $f(x)$ via såkalt finite differ-

ences. Når du lære om Taylor utvikling i matematikken vil du kunne vise at (central differences)

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

samt at

$$f'(x) \approx \frac{f(x+h/2) - f(x-h/2)}{h}$$

hvor h er skritt lengden (eller Δx). Mer informasjon om dette finner du f.eks her.

Vi vil nå skrive to funksjoner som vil kunne bergene de deriverte når kan kjenner funksjonen $f(x)$ for alle argumenter. For steglengden vil vi bruke en liten default verdi

[40]:
```
def diff1(f,x,h=1E-6):
    """Beregner en approksimasjon for den 1. deriverte av funksjonen f(x) ved␣
 ↪hjelp av sentral differansen.

    Args:
        f (): funksjon
        x (float eller dnarray):
        h (float, optional): steglengden. Defaults to 1E-6.

    Returns:
        float eller dnarray: den 1. deriverte av funksjonen f
    """
    res =  ( f(x+h/2) - f(x-h/2) ) / h
    return res
# end function


def diff2(f,x,h=1E-6):
    """Beregner en approksimasjon for den 2. deriverte av funksjonen f(x) ved␣
 ↪hjelp av sentral differansen.

    Args:
        f (): funksjon
        x (float eller dnarray):
        h (float, optional): steglengden. Defaults to 1E-6.

    Returns:
        float eller dnarray: den 2. deriverte av funksjonen f
    """
    res =  ( f(x+h) - 2*f(x) + f(x-h) ) / h**2
    return res
# end function
```

Nå skal vi teste disse funksjonen. La oss starte med den enkle funksjonen $f(x) = x^2$, slik at vi trivielt vet hva svaret skal være.

[ ]:
```
def ftest(x):
    return x*x
# end function
```

[49]:
```
N = 100
x = np.linspace(-2,2,N)

fx = ftest(x)
dfdx = diff1( ftest, x)
df2dx2 = diff2( ftest, x)
```

[ ]:
```
# plotting the results
import matplotlib.pyplot as plt

plt.plot(x,fx,label="$f(x)$")
plt.plot(x,dfdx,label="$f'(x)$")
plt.plot(x,df2dx2,label="$f''(x)$")

plt.xlabel("$x$")

plt.legend()
plt.show()
```

Dette ser da ganske så bra ut!

Hva om vi bruker *np.cos* som vår funksjon.

[ ]:
```
# Vår fuksjon
func = np.cos


N = 100
x = np.linspace(0,2*np.pi,N)

# Beregner deriverte
fx = func(x)
dfdx = diff1( func, x)
df2dx2 = diff2( func, x)

# Plotter resultatene
plt.plot(x,fx,label="$f(x)$")
plt.plot(x,dfdx,label="$f'(x)$")
plt.plot(x,df2dx2,label="$f''(x)$")

plt.xlabel("$x$")

plt.legend()
```

```
plt.show()
```

La oss også forsøke dette for en noe mer komplisert egendefinert funksjon.

[ ]:
```
def func(x):
    res = np.exp(-x/5) * np.sin(x)
    return res
# end function


N = 100
x = np.linspace(0,10*np.pi,N)

# Beregner deriverte
fx = func(x)
dfdx = diff1( func, x)
df2dx2 = diff2( func, x)

# Plotter resultatene
plt.plot(x,fx,label="$f(x)$")
plt.plot(x,dfdx,label="$f'(x)$")
plt.plot(x,df2dx2,label="$f''(x)$")

plt.xlabel("$x$")

plt.legend()
plt.show()
```

1.4 Lambda funksjon

Lambda functions er ikke noe annet en enkel og kompakt måte å definere en fusjon som en one-liner. For definere fuksjonen $f(x) = x^2$ som en lambda funksjon vi gjør følgende

[63]:
```
f = lambda x: x*x
```

Resten blir som før!

[ ]:
```
N = 100
x = np.linspace(-1,1,N)

# Beregner deriverte
fx = f(x)
dfdx = diff1( f, x)
df2dx2 = diff2( f, x)

# Plotter resultatene
plt.plot(x,fx,label="$f(x)$")
plt.plot(x,dfdx,label="$f'(x)$")
```

```
plt.plot(x,df2dx2,label="$f''(x)$")

plt.xlabel("$x$")

plt.legend()
plt.show()
```

Om man vil man man putte lambda funksjonen rett inn i argumentet til f.eks. diff1

[ ]:
```
# plt.plot(x,dfdx,label="$f'(x)$")
plt.plot(x, diff1( lambda x: x*x, x)  ,label="$f'(x)$")
```

Vi kan også ha flere argumenter om vi vil. De bli i så fall satt inn mellom lambda og : som vist nedenfor

[ ]:
```
g = lambda x, y: x*y

print(" g(2,4) = ", g(2,4) )
```

1.5 Applications

Hva kan man så bruke dette til? Vi vil her beskrive hvordan denne teknikken kan brukes til

- å finne nullpunkter av en funksjon ved hjelp av interval halveringsmetoden (en: bisectioning)
- Newtons metode for å løse ikke+lineære likninger

1.5.1 Interval halveringsmetoden

[80]:
```
def bisection(f,a,b,tol= 1e-3):

    if f(a)*f(b) > 0:
        print(f"No roots or more than one root in [{a},{b}]")
        return
    # end if

    m = (a+b)/2
    while abs(f(m)) > tol:
        if f(a)*f(m) < 0:
            b = m
        else:
            a = m
        m = (a+b)/2
        # end if
    #end while
    return m
# end function
```

[ ]:
```
# Test code for bisection
#call the method for f(x)= x**2-4*x+exp(-x)
```

```
f = lambda x: x**2-4*x+np.exp(-x)

a = -0.5
b = 1.

sol = bisection(f,a,b,1e-6)
print(f"x = {sol:g} is an approximate root, f({sol:g}) = {f(sol):g}")
```

Not surprisingly scipy also offers the same function, so let us see what this gives!

[ ]:
```
import scipy as sp

help(sp.optimize.bisect)
```

[ ]:
```
import scipy as sp

a = -0.5
b = 1
sol = sp.optimize.bisect(f,a,b)

print(f"x = {sol:g} is an approximate root, f({sol:g}) = {f(sol):g}")
```

1.5.2 Newtons method

Interval halverings metoden er ikke spesielt effektiv. Et mer effektivt alternativ for løsingen av ikke-lineære likninger er Newtons metode. Den er også kjent som Newton–Raphson metoden. Denne metoden starter fra et *initial guess* som vi vil kalle x_0, og så iterer den seg frem til hva som forhåpenligvis er en løsning via

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \qquad n = 1, 2, 3 \ldots .$$

Hvordan denne likningen utledes kan du se her fra.

Implementasjonen av denne metoden kan derfor skrives som følger:

[ ]:
```
def newton_raphson(f, dfdx, x0, tol= 1e-3):
    """Newton-Raphson iterasjon for å bergene nullpuntet for en funksjon f(x)

    Args:
        f (callable): funksjon f(x) som man søker nullpunktet for
        dfdx (callable): funksjon som returnerer f'(x)
        x0 (skalar): initielle gjetning
        tol (skalar, optional): toleransen. Defaults to 1e-3.

    Returns:
        skalar: estimat for nullpunktet
    """
    f0 = f(x0)
    while abs(f0) > tol:
```

```
        x1 = x0 - f0/dfdx(x0)
        x0 = x1
        f0 = f(x0)
    return x0
# end function
```

Tester denne rutine for å finne nullpuntet for funksjonen

$$f(x) = x^2 - 4x + \exp(-x).$$

Den deriverte av denne funksjonen er

$$f'(x) = 2x - 4 - \exp(-x).$$

Vi lager først to lambda-funksjoner i Python som definerer disse, og deretter skal vi bruke Newton-Raphson til å finne nullpuntet for $f(x)$.

[88]:
```
f = lambda x: x*x -4*x +np.exp(-x)

dfdx = lambda x: 2*x - 4 - np.exp(-x)
```

Hva skal vi bruke som vårt initial guess? La oss plotte funksjonen først for å se.

[ ]:
```
x=np.linspace(-2,5)
plt.plot(x,f(x))
plt.plot(x,0*x,'--') # Null linjen
plt.show()
```

Man ser at det er to nullpunkter så vi kan bruke, f.eks. $x_0 = -0.5$ for det første, og $x_0 = 3$ for det andre. [Merk deg hvordan hjelpetrenger opptrer for vår funksjon når musepekeren er over funsjonsnavnet]

[ ]:
```
# Første nullpunkt
x0 = -0.5
sol = newton_raphson(f,dfdx,x0)

print(f"x = {sol:g} is an approximate root, f({sol:g}) = {f(sol):g}")
```

OPPGAVE: Forsøk å endre toleransen. Hva ser du for det beregnede nullpunktet?

[ ]:
```
# Andre nullpunkt
x0 = 3.
sol = newton_raphson(f,dfdx,x0)

print(f"x = {sol:g} is an approximate root, f({sol:g}) = {f(sol):g}")
```

Merk deg at du vil konvergere til ett av de to nullpunktene avhengig av verdien du bruker for x_0. Prøv dette.

Et google søk, f.eks. vil gi deg at også denne metoden finnes i scipy.optimize.

[ ]:
```
help(sp.optimize.newton)
```

[ ]:
```
x0 = -0.5
sol = sp.optimize.newton(f,x0,dfdx)

print(f"x = {sol:g} is an approximate root, f({sol:g}) = {f(sol):g}")
```

Merk deg av denne rutinen ikke trenger den deriverte strengt tatt.....

[ ]:
```
x0 = -0.5
sol = sp.optimize.newton(f,x0)

print(f"x = {sol:g} is an approximate root, f({sol:g}) = {f(sol):g}")
```

1.6 Testing av kode

Når man utvikler kode er det god praksis å skrive tester. Slike tester kan være del av koden, eller man kan skrive separate *test funksjoner*. For å illustrere det test funksjoner, vil vi se på følgende eksempel

[131]:
```
def double(x):      # En tilfeldig funksjon
    return 2*x
# end function

def test_double():              # associated test function
    x = 4                       # some chosen x value
    expected = 8                # expected result from double(x)
    computed = double(x)
    success = ( computed == expected ) # Boolean value: test passed?
    msg = f"computed {computed}, expected {expected}"
    assert success, msg
# end function


test_double()
```

Her brukes *assert* som skriver ut noe bare om success er False. For å se hvordan dette oppfører seg kan du f.eks. omdefinere dobble til å returnere $3x$ istedet for $2x$. Merk deg at en test funksjon typisk ikke returnerer noe uten at testen ikke er oppfylt. Du kan også teste flere ting i en og samme test fuksjon.

Syntaxen for assert er: **assert "logisk-test", "melding"**

[137]:
```
def f(x):
    if 0 <= x <= np.pi:
        return np.sin(x)
    else:
        return 0
# end function
```

```
def test_f():
    x1, exp1 = -1.0, 0.0
    x2, exp2 = np.pi/2, 1.0
    x3, exp3 = 3.5, 0.0
    tol = 1e-10
    assert abs(f(x1)-exp1) < tol, f"Failed for x = {x1}"
    assert abs(f(x2)-exp2) < tol, f"Failed for x = {x2}"
    assert abs(f(x3)-exp3) < tol, f"Failed for x = {x3}"
# end function


# denne er lik test_f funksjonen
def test_2_f():
    x_vals = [-1, np.pi/2, 3.5]
    exp_vals = [0.0, 1.0, 0.0]
    tol = 1e-10
    for x, exp in zip(x_vals, exp_vals):
        assert abs(f(x)-exp) < tol, \
            f"Failed for x = {x}, expected {exp}, but got {f(x)}"
    # end for
# end function


# La oss kjøre testene (ingen output forventet)
test_f()
test_2_f()
```

Vi skal nå se på en annen nyttig konstruksjon når man skriver kode. La oss se på funksjonen

$$\text{sinc}(x) = \frac{\sin(x)}{x}.$$

[ ]:
```
def sinc(x):
    res = np.sin(x)/x
    return res
# end function
```

[ ]:
```
# Hvordan ser denne funksjonen ut?
x = np.linspace(-20,20,100)
plt.plot(x,sinc(x))
```

[ ]:
```
# Hva skjer i dette tilfellet....
sinc(0.)
```

La oss redefiner vår sinc(x) funksjon!

[ ]:
```
def sinc2(x):
    if (np.abs(x))>0:
        res = np.sin(x)/x
    else:
        res=1.
    # end if
    return res
# end function

s = sinc2(0.)
print( f" sinc(0) = {s}" )
```