

## Kap 3: Funksjoner og if-tester

Ole Christian Lingjærde, Institutt for Informatikk, UiO

2-5 September, 2019 (Del 2 av 2)

- **Lister:** kan brukes til å lage 1D-, 2D- og 3D-tabeller
- **Funksjoner:** brukes til å splitte programmer i mindre biter
- **Kall:** kalle på funksjon = anvende funksjonen
- **Metoder:** funksjoner som kalles med `.`-notasjon
- **Argumenter og returverdier:** input og output
- **Navngitte argumenter:** i kallet `f(x=0.5)` er `x` navngitt
- **Lokale variabler:** variabler definert inni funksjoner er lokale

- Mer om funksjoner
- If-tester
- Assert-tester
- Testfunksjoner

# Kort repetisjon av funksjoner

## Definere funksjoner

```
def skrivut():  
    print("Pi is approximately 3.1415926")  
  
def g(x,y):  
    return x**2 + y**2
```

## Bruke funksjoner

```
skrivut()  
print(g(0.5, 3.5))
```

# Eksempel 1

## Beregne gjennomsnittsverdi

```
# Definer funksjon
def snitt(a):
    s = 0
    for e in a:
        s = s + e
    return s/len(a)

# Prøv på en liste:
x = [3.6, 7.2, 9.3, -5.6, 0.1]
print("Gjennomsnittsverdien er %g" % snitt(x))

# Prøv på en annen liste:
y = [1, 1, 2, 2, 3, 3]
print("Gjennomsnittsverdien er %g" % snitt(y))
```

## Kjøreresultat

```
> python eksempel1.py
Gjennomsnittsverdien er 2.92
Gjennomsnittsverdien er 2
```

### Skrive en melding

```
# Definer funksjon
def skrivMelding(navn, melding):
    s = "Gratulerer %s, du har vunnet %s." % (navn, melding)
    print(s)

# Prøv funksjonen:
skrivMelding("Mia", "100 kr")
skrivMelding("Jon", "en million kroner")
```

### Kjørerresultat

```
> python eksempel2.py
Gratulerer Mia, du har vunnet 100 kr.
Gratulerer Jon, du har vunnet en million kroner.
```

## Eksempel 3

### Beregne indreprodukt

```
# Definer funksjon
def innerprod(x, y):
    v = 0
    for i in range(len(x)):
        v += x[i]*y[i]
    return v

# Prøv funksjonen:
x = [1, 2, 3, 4]
y = [0, 1, 4, 2]
xy = innerprod(x,y)
print("Indreproduktet av x og y er %5.2f" % xy)
```

### Kjørereresultat

```
> python eksempel3.py
Indreproduktet av x og y er 22.00
```

## Eksempel 4

Beregne  $n! = 1 \cdot 2 \cdot 3 \cdots n$

```
# Definer funksjon
def factorial(n):
    v = 1
    for i in range(1,n+1):
        v = v * i
    return v

# Prøv funksjonen:
for n in range(1,8):
    print("Factorial of %d = %d" % (n, factorial(n)))
```

Kjøreresultat

```
> python eksempel4.py
Factorial of 1 = 1
Factorial of 2 = 2
Factorial of 3 = 6
Factorial of 4 = 24
Factorial of 5 = 120
Factorial of 6 = 720
Factorial of 7 = 5040
```



Variabler definert *utenfor* funksjoner kalles globale variabler. De er også synlige inni funksjoner.

Eksempel:

```
def skrivut():  
    print("alpha = %g" % alpha)  
  
alpha = 10  
skrivut()           # Utskrift: alpha = 10
```

Variabler definert i funksjoner er lokale variabler. De er ikke synlige utenfor funksjonen og eksisterer bare når funksjonen utføres.

Tilordninger inni funksjoner lager lokale variabler. Eksempel:

```
def skrivut():  
    alpha = 0.6323          # LOKAL variabel  
    print("alpha = %g" % alpha)  
  
alpha = 10                 # GLOBAL variabel  
skrivut()                  # Utskrift: alpha = 0.6323  
print("alpha = %g" % alpha) # Utskrift: alpha = 10
```

# Lokale variabler maskerer globale variabler med samme navn

Hvis en lokal og en global variabel har samme navn, vil bare den lokale variabelen være synlig inni funksjonen.

Eksempel:

```
def g(t):  
    alpha = 1.0           # LOKAL variabel  
    beta = 2.0           # LOKAL variabel  
    return alpha + beta*t  
  
print(g(1))              # Skriver ut '3.0'  
alpha = 100000          # GLOBAL variabel  
print(g(1))              # Skriver ut '3.0'
```

Her blir verdien `alpha = 100000` aldri brukt.

## Endre globale variablers verdi i en funksjon

Det er ikke helt rett frem å endre verdien til en global variabel når vi er inni en funksjon. Eksempel:

```
def f():  
    x = 5  
    print(x)  
  
x = 10  
print(f()) # Skriver ut '5'  
print(x)  # Skriver ut '10'
```

Tilordningen `x=5` inni funksjonen endrer ikke den globale variabelen med samme navn, den lager en ny lokal variabel som settes lik 5.

# Endre globale variablers verdi i en funksjon

Hvis du virkelig ønsker å endre en global variablers verdi fra innsiden av en funksjon, må du deklare variabelen som *global*. Det er sjelden behov for å gjøre dette.

Eksempel:

```
def f():  
    global x          # Denne sier: ikke lag noen lokal variabel x  
    x = 5            # Nå endrer vi den globale variabelen x  
    return x  
  
x = 10  
print(f())          # Skriver ut '5'  
print(x)            # Skriver ut '5'
```

## Et litt større eksempel

Denne funksjonen approksimerer  $\ln x$  for  $x \geq 2$ :

$$L(x, n) = \sum_{i=1}^n \frac{1}{i} \left( \frac{x-1}{x} \right)^i$$

Eksempel:  $\ln 6 = 1.791759\dots$ , mens

$$L(6, 1) = 0.8333333$$

$$L(6, 2) = 1.180556$$

$$L(6, 3) = 1.373457$$

$$L(6, 4) = 1.49402$$

$$L(6, 5) = 1.574396$$

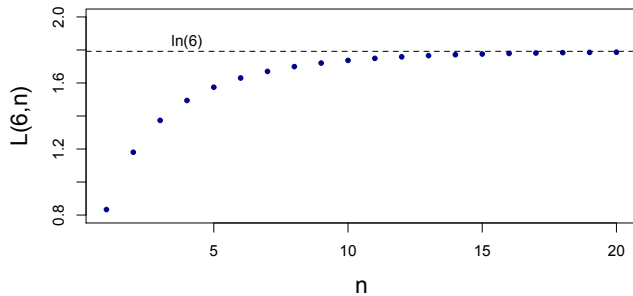
...

$$L(6, 20) = 1.786568$$

...

$$L(6, 100) = 1.791759$$

# Visuelt: $L(6,n)$ for $n=1,2,\dots,20$



Python-funksjon som implementerer  $L(x, n)$ :

```
def L(x,n):  
    s = 0  
    for i in range(1, n+1):  
        s += (1.0/i)*((x-1.0)/x)**i  
    return s
```



## Hvor forskjellig er $L(5, n)$ fra $\ln 6$ ?

- Anta at vi ønsker å finne  $\ln 6$  med stor nøyaktighet
- Hvor stor må vi velge  $n$ ?
- Visuelt ser vi at  $n > 20$  gir ganske nøyaktig svar
- Kan vi si noe mer presis om estimeringsfeilen?

To forslag:

- Regne ut feilen  $\ln(6) - L(5; n)$ .
- Regne ut første ledd i summen som *ikke* er tatt med

Løsning 1 er best, men da må vi regne ut  $\ln(..)$ . Hvis vi skal unngå å bruke fasitsvaret, er løsning 2 et alternativ.

## Utvidelse av $L(x,n)$

```
from math import log

def L(x,n):
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*((x-1.0)/x)**i

    feil1 = log(x) - s
    feil2 = (1.0/(n+1))*((x-1.0)/x)**(n+1)

    return s, feil1, feil2

# Typisk kall:
x = 1.2; n = 100
value, feil1, feil2 = L(x, n)
```

## Default-verdier for argumenter

Når vi definerer funksjoner kan vi oppgi default-verdier for argumentene. Eksempel:

```
def skrivut(x, y, z=1, w=2.5):  
    print(x, y, z, w)
```

# Kall på funksjoner hvor argumenter har default-verdier

```
def skrivut(x, y, z=0, t=99):  
    print (x, y, z, t)  
  
# Benytt default-verdiene for z og t:  
skrivut('Hei', 3) # Utskrift: Hei 3 0 99  
  
# Benytt default-verdien for t:  
skrivut('Hei', 3, z='XX') # Utskrift: Hei 3 XX 99  
  
# Benytt default-verdien for z:  
skrivut('Hei', 3, t='YY') # Utskrift: Hei 3 0 YY  
  
# Ikke benytt noen default-verdier:  
skrivut('Hei', 3, z='XX', t='YY') # Utskrift: Hei 3 XX YY
```

# Quiz 1

Anta at vi har definert en funksjon

```
def h(x, y, z=0):  
    import math  
    res = x * math.sin(y) + z  
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0)  
r = h(0, 1)  
r = h(0, 1, 2)  
r = h(x=0, 1, 2)  
r = h(0, y=1)  
r = h(0, 1, z=3)  
r = h(0, 0, x=0)  
r = h(z=0, x=1)  
r = h(z=0, x=1, y=2)
```

Anta at vi har definert en funksjon

```
def h(x, y, z=0):  
    import math  
    res = x * math.sin(y) + z  
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0) # Ikke lovlig (mangler y)  
r = h(0, 1)  
r = h(0, 1, 2)  
r = h(x=0, 1, 2)  
r = h(0, y=1)  
r = h(0, 1, z=3)  
r = h(0, 0, x=0)  
r = h(z=0, x=1)  
r = h(z=0, x=1, y=2)
```

Anta at vi har definert en funksjon

```
def h(x, y, z=0):  
    import math  
    res = x * math.sin(y) + z  
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0)                                # Ikke lovlig  
r = h(0, 1)                              # Lovlig  
r = h(0, 1, 2)  
r = h(x=0, 1, 2)  
r = h(0, y=1)  
r = h(0, 1, z=3)  
r = h(0, 0, x=0)  
r = h(z=0, x=1)  
r = h(z=0, x=1, y=2)
```

Anta at vi har definert en funksjon

```
def h(x, y, z=0):  
    import math  
    res = x * math.sin(y) + z  
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0)                # Ikke lovlig  
r = h(0, 1)             # Lovlig  
r = h(0, 1, 2)         # Lovlig  
r = h(x=0, 1, 2)  
r = h(0, y=1)  
r = h(0, 1, z=3)  
r = h(0, 0, x=0)  
r = h(z=0, x=1)  
r = h(z=0, x=1, y=2)
```



Anta at vi har definert en funksjon

```
def h(x, y, z=0):  
    import math  
    res = x * math.sin(y) + z  
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0) # Ikke lovlig  
r = h(0, 1) # Lovlig  
r = h(0, 1, 2) # Lovlig  
r = h(x=0, 1, 2) # Ikke lovlig (navngitte må komme sist)  
r = h(0, y=1)  
r = h(0, 1, z=3)  
r = h(0, 0, x=0)  
r = h(z=0, x=1)  
r = h(z=0, x=1, y=2)
```

Anta at vi har definert en funksjon

```
def h(x, y, z=0):  
    import math  
    res = x * math.sin(y) + z  
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0)                # Ikke lovlig  
r = h(0, 1)            # Lovlig  
r = h(0, 1, 2)        # Lovlig  
r = h(x=0, 1, 2)      # Ikke lovlig  
r = h(0, y=1)         # Lovlig  
r = h(0, 1, z=3)      # Lovlig  
r = h(0, 0, x=0)      # Lovlig  
r = h(z=0, x=1)       # Lovlig  
r = h(z=0, x=1, y=2)  # Lovlig
```

Anta at vi har definert en funksjon

```
def h(x, y, z=0):  
    import math  
    res = x * math.sin(y) + z  
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0) # Ikke lovlig  
r = h(0, 1) # Lovlig  
r = h(0, 1, 2) # Lovlig  
r = h(x=0, 1, 2) # Ikke lovlig  
r = h(0, y=1) # Lovlig  
r = h(0, 1, z=3) # Lovlig  
r = h(0, 0, x=0)  
r = h(z=0, x=1)  
r = h(z=0, x=1, y=2)
```

Anta at vi har definert en funksjon

```
def h(x, y, z=0):  
    import math  
    res = x * math.sin(y) + z  
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0) # Ikke lovlig  
r = h(0, 1) # Lovlig  
r = h(0, 1, 2) # Lovlig  
r = h(x=0, 1, 2) # Ikke lovlig  
r = h(0, y=1) # Lovlig  
r = h(0, 1, z=3) # Lovlig  
r = h(0, 0, x=0) # Ikke lovlig (x angis to ganger)  
r = h(z=0, x=1)  
r = h(z=0, x=1, y=2)
```

Anta at vi har definert en funksjon

```
def h(x, y, z=0):  
    import math  
    res = x * math.sin(y) + z  
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0) # Ikke lovlig  
r = h(0, 1) # Lovlig  
r = h(0, 1, 2) # Lovlig  
r = h(x=0, 1, 2) # Ikke lovlig  
r = h(0, y=1) # Lovlig  
r = h(0, 1, z=3) # Lovlig  
r = h(0, 0, x=0) # Ikke lovlig  
r = h(z=0, x=1) # Ikke lovlig (y mangler)  
r = h(z=0, x=1, y=2)
```

Anta at vi har definert en funksjon

```
def h(x, y, z=0):  
    import math  
    res = x * math.sin(y) + z  
    return res
```

Hvilke av disse funksjonskallene er lovlige?

```
r = h(0) # Ikke lovlig  
r = h(0, 1) # Lovlig  
r = h(0, 1, 2) # Lovlig  
r = h(x=0, 1, 2) # Ikke lovlig  
r = h(0, y=1) # Lovlig  
r = h(0, 1, z=3) # Lovlig  
r = h(0, 0, x=0) # Ikke lovlig  
r = h(z=0, x=1) # Ikke lovlig  
r = h(z=0, x=1, y=2) # Lovlig
```

Hva skrives ut her?

```
def skrivut(k):  
    x = k * 2  
    print('x = %g' % x)
```

```
x = 5  
print('x = %g' % x)  
skrivut(5)  
print('x = %g' % x)
```

Hva skrives ut her?

```
def skrivut(k):  
    x = k * 2  
    print('x = %g' % x)
```

```
x = 5  
print('x = %g' % x)      # x = 5  
skrivut(5)               # x = 10  
print('x = %g' % x)      # x = 5
```



# Mer om funksjoner - et eksempel

Anta at vi har en funksjon av  $t$ , med parametre  $A$ ,  $a$  og  $\omega$ :

$$f(t; A, a, \omega) = Ae^{-at} \sin(\omega t)$$

Mulige implementasjoner i Python:

```
from math import pi, exp, sin

def f(t, A=1, a=1, omega=2*pi):
    return A*exp(-a*t)*sin(omega*t)
```

Her har  $A$ ,  $a$  og  $\omega$  default-verdier. Det gir oss mange muligheter når vi kaller på funksjonen:

```
v1 = f(0.2) # Bare oppgi t
v2 = f(0.2, omega=1) # Endre omega
v3 = f(0.2, omega=1, A=2.5) # Endre omega og A
v4 = f(A=5, a=0.1, omega=1, t=1.3) # Endre alle parametre
v5 = f(0.2, 1, 2.5) # Endre A og a
```

# Funksjoner kan ha funksjoner som argumenter

I Python kan argumenter til en funksjon selv være funksjoner.

Eksempel: Hvis vi vet hvordan vi skal beregne  $f(x)$ , kan vi estimere den annenderiverte i et gitt punkt  $x$  med formelen:

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

Python-implementasjon:

```
def diff2(f, x, h=1E-6):  
    r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)  
    return r
```

Første argument til `diff2(.)` er her den funksjonen vi skal finne den annenderiverte til.

Funksjonen vi nettopp definerte, har nøkkelord-argumentet  $h=1E-6$ . Er det en grunn til å velge  $h = 0.000001$  i stedet for en lavere eller høyere verdi?

- Matematisk forventer vi at approksimasjonen blir bedre når  $h$  blir mindre.
- Når vi løser problemer numerisk, må vi også ta hensyn til avrundingsfeil.
- Noen numeriske problemer er mer følsomme for avrundingsfeil enn andre, så i praksis blir det ofte litt prøving og feiling for å finne en god  $h$ -verdi.

# Effekten av å endre h

For å studere effekten av å endre h skriver vi et lite program:

```
def g(t):  
    return t**(-6)  
  
# Beregn g''(t) for mindre og mindre verdier av h:  
for k in range(1,14):  
    h = 10**(-k)  
    print ('h=%.0e: %.5f' % (h, diff2(g, 1, h)))
```

Output ( $g''(1) = 42$ )

```
h=1e-01: 44.61504  
h=1e-02: 42.02521  
h=1e-03: 42.00025  
h=1e-04: 42.00000  
h=1e-05: 41.99999  
h=1e-06: 42.00074  
h=1e-07: 41.94423  
h=1e-08: 47.73959  
h=1e-09: -666.13381  
h=1e-10: 0.00000  
h=1e-11: 0.00000  
h=1e-12: -666133814.77509  
h=1e-13: 66613381477.50939
```

# For små $h$ -verdier dominerer avrundingsfeil

For  $h < 10^{-8}$  er resultatene helt feil!

- **Problem 1:** for små  $h$  subtraherer vi tall som er nesten like and dette gir opphav til avrundingsfeil.
- **Problem 2:** for små  $h$  deler vi avrundingsfeilen på et veldig lite tall ( $h^2$ ), og dette forsterker feilen.

Mulig løsning: bruke desimaltall med flere sifre

- Python har en (langsom) flyttalls datatype (`decimal.Decimal`) hvor antall sifre er vilkårlig stort.
- Bruk av 25 sifre gir nøyaktige resultater for  $h \leq 10^{-13}$

I praksis er det sjelden behov for høyere nøyaktighet.

# Funksjoner vs hovedprogram

Hovedprogrammet er den delen av programmet som ikke ligger inni noen funksjon. Generelt:

- Progameksekveringen starter med første programsetning i hovedprogrammet og fortsetter linje for linje, fra topp til bunn.
- Funksjoner utføres bare når man kaller på dem.

**Merk:** funksjoner kan kalles fra hovedprogrammet *eller* fra en funksjon. Dette kan noen ganger føre til lange "kjeder" av funksjonskall.

# Lambda-funksjoner

Noen ganger trenger vi å lage funksjoner som bare beregner enkle uttrykk. Da er *lambda-funksjoner* ofte et nyttig alternativ til vanlige funksjonsdefinisjoner.

Eksempel: funksjonen

```
def f(x,y):  
    return x**2 - y**2
```

kan defineres på én linje med lambda-konstruksjonen:

```
f = lambda x, y: x**2 - y**2
```

Lambdafunksjoner kan brukes direkte som argumenter i funksjoner:

```
z = g(lambda x, y: x**2 - y**2, 4)
```

# Funksjoner bør dokumenteres

For å legge til en kort beskrivelse av en funksjon lager vi en *doc string* som plasseres rett etter funksjons-headeren og inni triple anførselstegn.

Eksempler:

```
def C2F(C):  
    """Konverter Celsiusgrader (C) til Fahrenheit."""  
    return (9.0/5)*C + 32  
  
def line(x0, y0, x1, y1):  
    """  
    Beregn koeffisientene a og b i uttrykket for en rett  
    linje  $y = a*x + b$  som passerer gjennom  $(x_0, y_0)$  og  $(x_1, y_1)$   
  
     $x_0, y_0$ : første punkt (floats).  
     $x_1, y_1$ : andre punkt (floats).  
    return: a, b (floats) for linjen ( $y=a*x+b$ ).  
    """  
    a = (y1 - y0)/(x1 - x0)  
    b = y0 - a*x0  
    return a, b
```



If-tester gjør det mulig å lage forgreninger i programkjøringen, dvs at ulike handlinger utføres under ulike betingelser. Eksempel:

$$f(x) = \begin{cases} \sin x, & 0 \leq x \leq \pi \\ 0, & \text{otherwise} \end{cases}$$

For å implementere  $f$  i Python, trenger vi å teste hvilken verdi  $x$  har for å vite hvilken handling som skal utføres:

```
from math import sin, pi

def f(x):
    if 0 <= x <= pi:
        return sin(x)
    else:
        return 0
```

# Generell form på if-testen

## Type 1 (if)

```
if betingelse:  
    <setninger som eksekveres hvis betingelse==True>
```

## Type 2 (if-else)

```
if betingelse:  
    <setninger som eksekveres hvis betingelse==True>  
else:  
    <setninger som eksekveres hvis betingelse==False>
```

## Type 3 (if-elif-else)

```
if betingelse1:  
    <setninger>  
elif betingelse2:  
    <setninger>  
elif betingelse3:  
    <setninger>  
else:  
    <setninger>
```

# Eksempel 1

Stykkevis definert funksjon:

$$N(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \leq x < 1 \\ 2 - x, & 1 \leq x < 2 \\ 0, & x \geq 2 \end{cases}$$

Implementasjon med if-elif-else:

```
def N(x):  
    if x < 0:  
        return 0  
    elif 0 <= x < 1:  
        return x  
    elif 1 <= x < 2:  
        return 2 - x  
    elif x >= 2:  
        return 0
```

## Eksempel 2

Funksjon som teller hvor mange ganger s forekommer i a:

```
def count(s, a):  
    cnt = 0  
    for e in a:  
        if e == s:  
            cnt += 1  
    return cnt
```

Eksempel på bruk:

```
>>> count(5.3, [2.2, 6.6, 2.5, 5.3, 8.9, 5.3])  
>>> 2  
>>>  
>>> count('Anna', ['Ola', 'Karianne', 'Anna', 'Jens'])  
>>> 1  
>>>  
>>> count([1,2], [1, 5, [1,2], [1,2], 3])  
>>> 2
```

## Vanlig konstruksjon:

```
if betingelse:  
    variable = verdi1  
else:  
    variable = verdi2
```

## Mer kompakt syntaks med samme effekt:

```
variable = (verdi1 if betingelse else verdi2)
```

## Eksempel:

```
def f(x):  
    return (sin(x) if 0 <= x <= 2*pi else 0)
```

## assert: en spesiell form for test

Noen ganger ønsker man å stoppe programkjøringen og gi en feilmelding dersom en bestemt betingelse ikke er oppfylt. For dette formål har vi `assert`. Generell form:

```
assert betingelse, melding
```

Eksempel:

```
>>> x = 5
>>> assert x > 0, "x maa vaere positiv" # Ingenting skjer
>>> x = -5
>>> assert x > 0, "x maa vaere positiv" # Gir feilmelding
Traceback (most recent call last):

  File "<ipython-input-30-c680011d20e2>", line 1, in <module>
    assert x > 0, "x should be positive"

AssertionError: x maa vaere positiv
```

Anta at vi har laget en funksjon i Python som returnerer en verdi. Hvordan kan vi vite om funksjonen fungerer som den skal?

Enkel strategi: sjekk at funksjonen iallfall gir riktig svar på noen utvalgte eksempler!

## Teststrategi

- Anta at vi har laget en funksjon  $f(x)$  og at vi ønsker å vite at kallet  $y = f(x)$  gir riktig svar.
- Vi lager en *testfunksjon* i Python som kaller på  $f(x)$  med noen utvalgte verdier for  $x$  der vi vet hva output  $y$  skal være (vi har fasiten).
- Hvis output ikke stemmer med fasiten, skal testfunksjonen gi en feilmelding.

# Eksempel

```
# Funksjon som skal finne a[0]+a[k]+a[2k]+...
def finn_sum(a,k):
    res = sum([a[i] for i in range(0,len(a),k)])
    return res

# Vi lager en testfunksjon:
def test_finn_sum():
    """Testfunksjon for finn_sum."""
    a = [0,1,2,3,4,5]      # Eksempel på inputverdi for a
    k = 3                  # Eksempel på inputverdi for k
    expected = 3           # Hva output forventes å bli
    computed = sum3(a,k)   # Hva output faktisk er
    success = (computed == expected) # Fikk vi riktig svar?
    message = 'computed %s, expected %s' % (computed, expected)
    assert success, message

# Vi kaller på testfunksjonen:
test_finn_sum()
```



## Utvidelse til flere tester

```
# Funksjon som skal finne a[0]+a[k]+a[2k]+...
def finn_sum(a,k):
    res = sum([a[i] for i in range(0,len(a),3)])
    return res

# Vi lager en testfunksjon:
def test_finn_sum():
    """Testfunksjon for finn_sum."""
    tol = 1E-14
    k = 3
    inputs = [[6], [6,1], [6,1,2], [6,1,2,3]]
    answers = [6, 6, 6, 9]
    for a, expected in zip(inputs, answers):
        computed = finn_sum(a,k)
        message = '%s != %s' % (computed, expected)
        assert abs(expected - computed) < tol, message

# Vi kaller på testfunksjonen:
test_finn_sum()
```

Husk at zip(a, b) lager tupler (a[i],b[i]):

```
>>> zip(inputs, answers)
>>> [(6, 6), (6, 1), (6, 1, 2), (6, 1, 2, 3), (9, 9)]
```

# Krav til testfunksjoner

En testfunksjon vil *ikke* gi noe utskrift på skjermen hvis funksjonen som testes består alle testene. Hvis en av testene feiler, vil det komme en feilmelding (`AssertionError`).

## Regler for testfunksjoner:

- Hvis navnet på funksjonen som skal testes er `XXX` så skal navnet på testfunksjonen være `test_XXX`.
- En testfunksjon har ingen argumenter
- En testfunksjon må ha en `assert success, message` setning, hvor `success` er `True` hvis testen består og `False` ellers. Siste argument `message` kan droppes.

# Hvordan kjøres testfunksjonene?

- **Fra programmet:** Vi kan legge inn et kall på testfunksjonen i samme program som funksjonen er definert.
- **Fra ipython:** Hvis vi kjører Python interaktivt via ipython kan vi kalle på testfunksjonen derfra.
- **Fra kommandolinjen:** Hvis vi står på samme filkatalog (directory) som programfilen, kan vi gi kommandoen `pytest prog.py` for å kjøre alle testfunksjoner i programfilen `prog.py`.
- **Forenklet testing:** Vi kan gi kommandoen `pytest` (uten argumenter) for å kjøre alle testfunksjoner i alle Python-programmer med navn som starter på `test_`.

## Enhetstester

Testfunksjoner som tester en liten bit av et program (en funksjon) kalles ofte *enhetstester*. For å teste et helt program, må programmet passere alle enhetstestene.

# Sluttkommentar om testfunksjoner

- Å bevise at et program oppfører seg korrekt for alle tenkelige input er generelt svært vanskelig.
- Å vise at programmet oppfører seg korrekt for *noen* inputverdier er et skritt på veien og kan ofte være tilstrekkelig.
- Det at en suksessfull test ikke gir noe output kan være irriterende - men bruker du pytest så får du beskjed om hvilke testfunksjoner som er kjørt.

# Oppsummering

If-tester:

```
if x < 0:  
    value = -1  
elif x >= 0 and x <= 1:  
    value = x  
else:  
    value = 1
```

Egendefinerte funksjoner:

```
def quadratic_polynomial(x, a, b, c):  
    value = a*x*x + b*x + c  
    derivative = 2*a*x + b  
    return value, derivative  
  
p, dp = quadratic_polynomial(1, 2, 0.5, 1)  
p, dp = quadratic_polynomial(x=x, a=-4, b=0.5, c=0)
```

Argumenter med default-verdi må komme sist:

```
def f(x, A=1, a=1, w=pi):  
    return A*exp(-a*x)*sin(w*x)
```

## Et litt større oppsummerende eksempel

Verdien til det bestemte integralet

$$\int_a^b f(x)dx$$

kan approksimeres med *Simpsons regel*:

$$\int_a^b f(x)dx \approx \frac{b-a}{3n} \left( f(a) + f(b) + 4 \sum_{i=1}^{n/2} f(a + (2i-1)h) + 2 \sum_{i=1}^{n/2-1} f(a + 2ih) \right)$$

hvor  $n$  er et partall. Vi skal se hvordan dette kan implementeres i Python.

## Forenkling av formelen

Vi kan skrive formelen for Simpsons regel slik:

$$\int_a^b f(x)dx \approx \frac{b-a}{3n} (A + 4B + 2C)$$

hvor

$$A = f(a) + f(b)$$

$$B = \sum_{i=1}^{n/2} f(a + (2i - 1)h)$$

$$C = \sum_{i=1}^{n/2-1} f(a + 2ih)$$

Vi skal lage en Python-funksjon `simpson(f, a, b)` for å beregne det bestemte integralet av  $f(x)$  på intervallet  $[a, b]$  ved hjelp av formelen over.

```
def simpson(f, a, b, n=500):  
    h = (b-a)/float(n)  
    m = n/2  
  
    A = f(a) + f(b)  
  
    B = 0  
    for i in range(1, m+1):  
        B = B + f(a + (2*i-1)*h)  
  
    C = 0  
    for i in range(1, m):  
        C = C + f(a + 2*i*h)  
  
    D = (b-a)/(3*n)*(A + 4*B + 2*C)  
  
    return D
```



## Utvidelse: sjekk at input er korrekt

```
def simpson(f, a, b, n=500):
    if a > b:
        print("Feil! Du maa velge a <= b")
        return None

    if n%2 != 0:
        print("Feil! Du maa velge n som et partall")
        return None

    h = (b-a)/float(n)
    m = n/2
    A = f(a) + f(b)
    B = 0
    for i in range(1, m+1):
        B = B + f(a + (2*i-1)*h)
    C = 0
    for i in range(1, m):
        C = C + f(a + 2*i*h)
    D = (b-a)/((3*n)*(A + 4*B + 2*C))

    return D
```

# Fullt program med anvendelse

```
from math import sin, pi

# Definer simpson-funksjonen
def simpson(f, a, b, n=500):
    <koden på forrige slide>

# Definer funksjonen som skal integreres
def f(x):
    return (3./2)*sin(x)**3

# Definer funksjonen som anvender simpson
def application():
    print("Integral of 1.5*sin^3 from 0 to pi:")
    print("  n          y          feil")
    for n in [2, 6, 12, 100, 500]:
        y = simpson(f, 0, pi, n)
        print ("%3d %18.15f %9.2E" % (n, y, 2-y))

# Anvend simpson
application()
```

Integral of  $1.5 \sin^3$  from 0 to pi:

n	y	feil
2	3.141592653589793	-1.14E+00
6	1.989171700583579	1.08E-02
12	1.999489233010781	5.11E-04
100	1.999999902476350	9.75E-08
500	1.999999999844138	1.56E-10

# Verifikasjon med testfunksjon

Annengradspolynomer integreres eksakt med Simpsons regel.

```
def test_simpson():  
    # Eksempel på inputverdier:  
    g = lambda x: 3*x**2-7*x+2.5  
    a = 1.5; b = 2.0; n = 8  
  
    # Forventet output:  
    G = lambda x: x**3 - 3.5*x**2 + 2.5*x # Integral av g  
    expected = G(b) - G(a)  
  
    # Beregnet output:  
    actual = simpson(g, a, b, n)  
  
    # Feilmelding hvis beregnet output ulik forventet:  
    suksess = abs(expected - actual) < 1E-14  
    melding = "Faktisk output er ulik forventet output!"  
    assert suksess, melding
```

Vi prøver testfunksjonen:

```
Terminal> pytest simpson.py
```

```
...
```

```
Ran 1 test in 0.005s
```

```
OK
```

## Exercise 3.20

*Write functions*

Three functions `hw1`, `hw2`, and `hw3` work as follows:

```
>>> print(hw1())
>>> Hello, World
>>>
>>> hw2()
>>> Hello, World
>>>
>>> print(hw3('Hello, ', 'World'))
>>> Hello, World
>>>
>>> print(hw3('Python ', 'function'))
>>> Python function
```

Write the three functions.

Filename: `hw_func`.

## Exercise 3.23

*Wrap a formula in a function*

Implement the formula (1.9) from Exercise 1.12 in a Python function with three arguments: `egg(M, To=20, Ty=70)`.

$$t = \frac{M^{2/3} c \rho^{1/3}}{K \pi^2 (4\pi/3)^{2/3}} \ln \left[ 0.76 \frac{T_0 - T_w}{T_y - T_w} \right].$$

The parameters  $\rho$ ,  $K$ ,  $c$ , and  $T_w$  can be set as local (constant) variables inside the function. Let  $t$  be returned from the function. Compute  $t$  for these conditions:

- Soft ( $T_y < 70$ ) and hard boiled ( $T_y > 70$ )
- Small ( $M = 47\text{g}$ ) and large ( $M = 67\text{g}$ ) egg
- Fridge ( $T_0 = 4\text{C}$ ) and hot room ( $T_0 = 25\text{C}$ ).

Filename: `egg_func`.

## Exercise 3.23 (solution)

```
from math import log, pi
def egg(M, T0=20, Ty=70):
    rho = 1.038; K = 5.4e-3; c = 3.7; Tw = 100
    A = M**(2.0/3) * c * rho**(1.0/3)
    B = (K*pi**2 * ((4*pi)/3)**(2.0/3)) * log(0.76*(T0-Tw)/(Ty-Tw))
    return A/B
```

*# For soft boiled eggs, yolk temp Ty should be <= 70, while for  
# hard boiled eggs Ty should reach 70. See Ex. 1.12 on page 46.  
# Here we use Ty=65 for soft boiled and Ty=80 for hard boiled egg.*

```
# Soft boiled, small, from fridge:
print(egg(47, T0=4, Ty=65))/60
# Soft boiled, large, from fridge:
print(egg(67, T0=4, Ty=65))/60
# Soft boiled, small, from hot room:
print(egg(47, T0=25, Ty=65))/60
# Soft boiled, large, from hot room:
print(egg(67, T0=25, Ty=65))/60
# Hard boiled, small, from fridge:
print(egg(47, T0=4, Ty=80))/60
# Hard boiled, large, from fridge:
print(egg(67, T0=4, Ty=80))/60
# Hard boiled, small, from hot room:
print(egg(47, T0=25, Ty=80))/60
# Hard boiled, large, from hot room:
print(egg(67, T0=25, Ty=80))/60
```

## Exercise 3.28

*Find the max and min elements in a list*

Given a list `a`, the `max` function in Python's standard library computes the largest element in `a`: `max(a)`. Similarly, `min(a)` returns the smallest element in `a`.

Write your own `max` and `min` functions.

*Hint:* Initialize a variable `max_elem` by the first element in the list, then visit all the remaining elements (`a[1:]`), compare each element to `max_elem`, and if greater, set `max_elem` equal to that element. Use a similar technique to compute the minimum element.

Filename: `maxmin_list`.