

Kap. 4: Funksjoner

Joakim Sundnes^{1,2}

Simula Research Laboratory¹

Universitetet i Oslo, Institutt for informatikk²

Aug 31, 2026

Plan for tirsdag 1. september

- Live-programmering, lister og løkker:
 - Opppg 2.15, 2.2, 2.16
- Nytt tema, funksjoner:
 - Parametre, argumenter
 - Returverdier
 - Lokale vs globale variabler
- Live-programmering:
 - Opppg 3.20, 3.23

- Live-programmering, funksjoner:
 - Oppg 3.28 ++
- Mer om funksjoner:
 - Funksjoner som argumenter til funksjoner
 - Lambda-funksjoner
 - doc-strenger
 - Rekursjon

Vi har brukt mange Python-funksjoner

Matematiske funksjoner:

```
from numpy import sin, log  
y = sin(x) * log(x)
```

Andre funksjoner:

```
n = len(somelist)  
integers = range(5, n, 2)
```

Funksjoner brukt med punktum-syntaks (kalt *metoder*):

```
C = [5, 10, 40, 45]  
i = C.index(10)           # resultat: i=1  
C.append(50)  
C.insert(2, 20)
```

Hva er en funksjon? Så langt har vi sett at vi putter noen objekter inn, og noen ganger får et objekt (resultat) ut av funksjoner. Nå er det på tide å skrive våre egne funksjoner!

Vi har brukt mange Python-funksjoner

Matematiske funksjoner:

```
from numpy import sin, log  
y = sin(x) * log(x)
```

Andre funksjoner:

```
n = len(somelist)  
integers = range(5, n, 2)
```

Funksjoner brukt med punktum-syntaks (kalt *metoder*):

```
C = [5, 10, 40, 45]  
i = C.index(10)           # resultat: i=1  
C.append(50)  
C.insert(2, 20)
```

Hva er en funksjon? Så langt har vi sett at vi putter noen objekter inn, og noen ganger får et objekt (resultat) ut av funksjoner. Nå er det på tide å skrive våre egne funksjoner!

Funksjoner er et av de viktigste verktøyene i programmering

- Funksjon = en gjenbrukbar kodeblokk som vi gir et navn og som kan utføres hvor vi vil i koden (ved å *kalle* på funksjonen)
- Funksjonen kan ta *input-objekter* (argumenter) og produsere output-objekter (returnere resultater)
- Funksjoner hjelper med å organisere programmer, gjør dem mer forståelige, kortere, gjenbrukbare, og lettere å utvide

Python-funksjon for å implementere en matematisk funksjon

Den matematiske funksjonen

$$F(C) = \frac{9}{5}C + 32$$

kan implementeres i Python slik:

```
def F(C):  
    return (9 / 5) * C + 32
```

Merk:

- Funksjoner starter med `def` etterfulgt av navnet på funksjonen og en liste med parametere/argumnter (her `C`) - denne linjen kalles gjerne funksjonens *signatur* eller *header*
- Inne i funksjonen: en innrykket blokk med setninger
- Nøkkelordet `return` "stopper" funksjonen og returnerer en verdi (vanligvis siste linje)

Funksjoner må kalles

En funksjon gjør ingenting før den blir kalt

```
def F(C):  
    return (9 / 5) * C + 32  
  
a = 10  
F1 = F(a) # kall  
temp = F(15.5) # kall  
print F(a+1) # kall  
sum_temp = F(10) + F(20) # to kall  
Fdegrees = [F(C) for C in [0, 20, 40]] # flere kall
```

Merk:

Kallet $F(C)$ produserer (returnerer) et float-objekt, som betyr at $F(C)$ blir erstattet av dette float-objektet. Vi kan derfor utføre kallet $F(C)$ overalt der en float kan brukes.

Funksjoner kan ta så mange argumenter du vil

Lag en Python-funksjon av den matematiske funksjonen

$$y(t) = v_0 t - \frac{1}{2} g t^2$$

```
def yfunc(t, v0):  
    g = 9.81  
    return v0 * t - 0.5 * g * t**2
```

eksempel på kall:

```
y = yfunc(0.1, 6)  
y = yfunc(0.1, v0 = 6)  
y = yfunc(t = 0.1, v0 = 6)  
y = yfunc(v0 = 6, t = 0.1)
```

Input til funksjoner - regel 1

- Input i en funksjonsdefinisjon kalles *parametre*
- Input i et funksjonskall kalles *argumenter*

Eksempel:

Parametre:

```
def f(x, y):  
    return x + y
```

Argumenter:

```
a = 0  
b = 1  
verdi = f(a, b)
```

Antallet argumenter i et funksjonskall må være det samme som antallet parametre i funksjonen.

Eksempel:

```
def f(x,y):  
    return x + y  
  
resultat = f(0)           #Ikke OK  
resultat = f(0, 1)        #OK  
resultat = f(0, 1, 2)     #Ikke OK
```

Vi kan gi standardverdier til parametre når vi definerer en funksjon. Da trenger vi ikke å oppgi verdier når funksjonen kalles.

Eksempel:

```
def f(x, y, z=2):  
    return x + y + z  
  
resultat = f(0)           #Ikke OK  
resultat = f(0, 1)        #OK  
resultat = f(0, 1, 2)     #OK
```

Merk:

parametre med standardverdier må stå sist i listen over parametre!

Eksempel:

```
def f(x, y=0, z=0):      #OK  
    print(x + y + z)
```

```
def f(x, y=0, z):        #Ikke OK  
    print(x + y + z)
```

```
def f(x=0, y, z):        #Ikke OK  
    print(x + y + z)
```

Vi kan navngi argumenter når vi kaller en funksjon. Disse kalles *nøkkelordsargumenter* (keyword arguments), og de må komme sist i argumentlisten.

Eksempel:

```
def f(x, y, z):  
    return x + y + z  
  
resultat = f(x=0, y=1, z=2) #OK  
resultat = f(0, y=1, z=2)  #OK  
resultat = f(0, z=2, y=1)  #OK  
resultat = f(z=2, 0, 1)    #Ikke OK
```

Funksjonsargumenter blir lokale variabler

```
def yfunc(t, v0):  
    g = 9.81  
    return v0 * t - 0.5 * g * t**2
```

```
v0 = 5  
t = 0.6  
y = yfunc(t, 3)
```

Lokale vs globale variabler

Når vi kaller `yfunc(t, 3)`, blir alle disse setningene faktisk utført:

```
t = 0.6 # oppretter en lokal variabel t  
v0 = 3  # lokal variabel v0  
g = 9.81 # lokal variabel  
return v0*t - 0.5*g*t**2
```

Inne i `yfunc` er `t`, `v0`, og `g` *lokale variabler*, ikke synlige utenfor `yfunc` og ødelagt etter `return`.

Utenfor `yfunc` (i hovedprogrammet), er `t`, `v0`, og `y` *globale variabler*, synlige overalt.

Funksjoner kan få tilgang til globale variabler

yfunc(t,v0)-funksjonen tok to argumenter, men vi kunne også ha implementert $y(t)$ som en funksjon av bare t :

```
def yfunc(t):  
    g = 9.81  
    return v0*t - 0.5*g*t**2
```

```
v0 = 5  
t = 0.6  
yfunc(t)
```

Merk: $v0$ og t (i hovedprogrammet) er globale variabler, mens t -en i `yfunc` er en lokal variabel.

Lokale variabler skjuler globale variabler med samme navn

Test dette:

```
def yfunc(t):  
    print(f"1. lokal t inne i yfunc: {t}")  
    g = 9.81  
    t = 0.1  
    print(f"2. lokal t inne i yfunc: {t}")  
    return v0 * t - 0.5 * g * t**2  
  
t = 0.6  
v0 = 2  
print(yfunc(t))  
print(f"1. global t: {t}")  
print(yfunc(0.3))  
print(f"2. global t: {t}")
```

Question

Hva blir skrevet ut?

Globale variabler *kan* endres hvis vi definerer dem som `global`

```
def yfunc(t):  
    g = 9.81  
    global v0    # nå kan v0 endres inni denne funksjonen  
    v0 = 9  
    return v0*t - 0.5*g*t**2  
  
v0 = 2    # global variabel  
print(f"1. v0: {v0}")  
print(yfunc(0.8))  
print(f"2. v0: {v0}")
```

Merk: det er veldig sjelden (aldri) nødvendig å bruke nøkkelordet `global`. Hvis man ønsker å endre på en global variabel er det bedre å la den være input til funksjonen og så returnere den nye verdien.

Funksjoner kan returnere flere verdier

Vi ønsker å beregne både $y(t)$ og $y'(t) = v_0 - gt$:

```
def yfunc(t, v0):  
    g = 9.81  
    y = v0*t - 0.5*g*t**2  
    dydt = v0 - g*t  
    return y, dydt
```

kall:

```
position, velocity = yfunc(0.6, 3)
```

- Objektene som skal returneres skilles med komma (egentlig returneres en *tuppel*).
- Variabler som skal tilordnes skilles med komma

```
>>> def f(x):  
>>> ... return x, x**2, x**4  
>>> ...  
>>> s = f(2)  
>>> s  
>>> (2, 4, 16)  
>>> type(s)  
>>> <type 'tuple'>  
>>> x, x2, x4 = f(2)    # samme syntaks som x, y = (obj1, obj2)
```

Eksempel: Funksjon som regner ut en sum

Funksjonen

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

er en tilnærming til $\ln(1+x)$ for en endelig n og $x \geq 1$.

Python-implementasjon av $L(x; n)$:

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

Vi kan også returnere feilen

Vi kan returnere mer, f.eks. det første leddet vi ikke tar med i summen, samt feilen $(\ln(1+x) - L(x; n))$:

```
def L2(x, n):
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1+x))**i
    value_of_sum = s
    first_neglected_term = (1.0/(n+1))*(x/(1+x))**(n+1)

    exact_error = log(1+x) - value_of_sum
    return value_of_sum, first_neglected_term, exact_error

# typisk kall:
x = 1.2; n = 100
value, approximate_error, exact_error = L2(x, n)
```

Funksjoner trenger ikke ha en return-setning

```
def somefunc(obj):  
    print(obj)  
  
return_value = somefunc(3.4)
```

Her får variabelen `return_value` verdien `None` fordi funksjoen ikke har en `return`-setning. Da setter Python automatisk inn `return None`.

Eksempel på en funksjon uten returverdi

Lag en tabell med $L(x; n)$ vs. $\ln(1+x)$:

```
def table(x):  
    print(f'x={x}, ln(1+x)={log(1+x)}')  
    for n in [1, 2, 10, 100, 500]:  
        value, next, error = L2(x, n)  
        print(f"n={n}, verdi = {value:g}, neste = {next:g}, feil = {er  
  
table(1)
```

- Ingen grunn til å returnere noe her - formålet er å skrive ut.
- Funksjoner som ikke returnerer noe kalles gjerne *prosedyrer*. I Python-programmer er ikke dette skillet veldig viktig.