Forelesning IN1900 – 7 September 2022

Ole Christian Lingjærde
Institutt for Informatikk, Universitetet i Oslo

Uke: 5 September – 11 September, 2022

Forrige forelesning på en slide

Lister: for å lagre flere verdier i ett dataobjekt

Funksjoner: navngi kodebiter slik at de kan brukes flere steder

Funksjonskall: anvende en funksjon

Argumenter og returverdier: input og output til funksjoner

Tema i dag

- Mer om funksjoner
- Assert-tester
- Testfunksjoner

Quiz 1

Virker dette programmet, og hva skriver det isåfall ut?

```
def skriv_ut:
    print("Hello world")

skriv_ut
```

Quiz 2

Virker dette programmet, og hva skriver det isåfall ut?

```
def skriv_ut():
    print("Hello world")

skriv_ut()
```

Quiz 3

Virker dette programmet, og hva skriver det isåfall ut?

```
def skriv_ut(tekst):
    print(tekst)

skriv_ut()
```

Quiz 4

Virker dette programmet, og hva skriver det isåfall ut?

```
def skriv_ut(tekst):
    print(tekst)

skriv_ut("Hello world")
```

Quiz 5

Virker dette programmet, og hva skriver det isåfall ut?

```
def regn_ut(x):
    y = x**2 + 2*x + 1

print(10)
```

Quiz 6

Virker dette programmet, og hva skriver det isåfall ut?

```
def regn_ut(x):
    y = x**2 + 2*x + 1
    return y

print(10)
```

Quiz 7

Virker dette programmet, og hva skriver det isåfall ut?

```
def f(x):
    y = x**2 + 2*x + 1
    return x,y

x0, y0 = f(10)
print(y0)
```

Quiz 8

Virker dette programmet, og hva skriver det isåfall ut?

```
def f(x):
    return x**2

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

print(g(2))
```

Quiz 9

Virker dette programmet, og hva skriver det isåfall ut?

```
def f(n):
    v = 1
    for i in range(2,n+1):
        v = v*i
    return v

print(f(4))
```

Litt teori om funksjoner

Vi vil nesten alltid ønske å overføre verdier til en funksjon når vi kaller på den.

Viktig å forstå mekanismene godt!

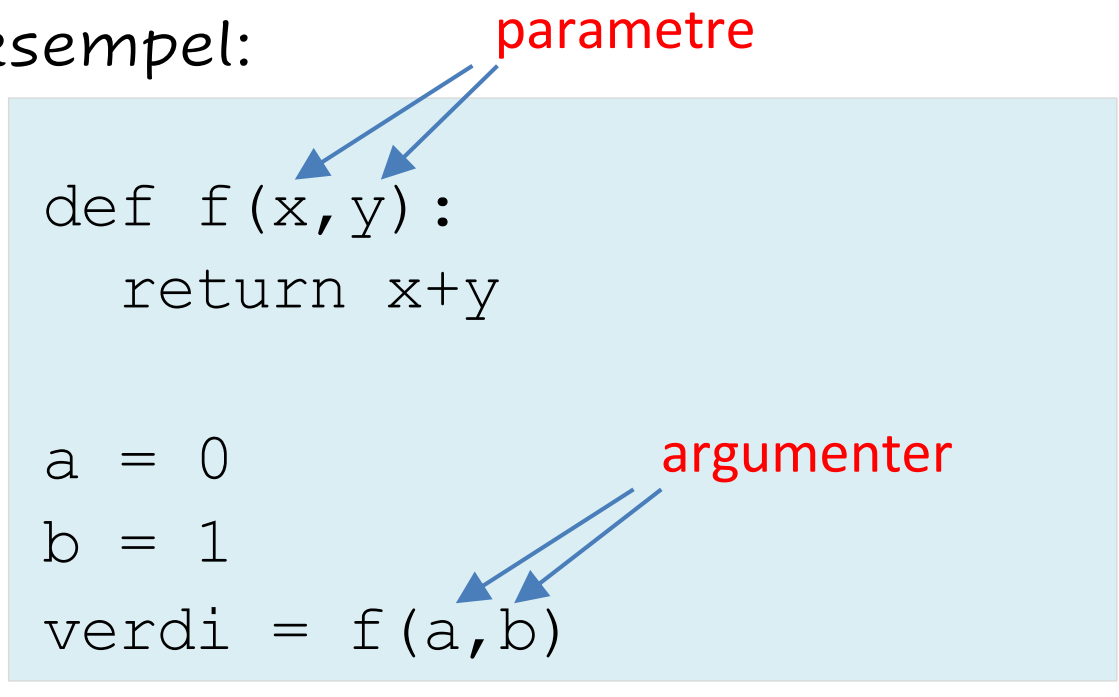
På de neste slidene går vi trinn for trinn gjennom de viktigste prinsippene.

Overføring av verdier til funksjoner

Regel 1:

- Inputs i en funksjonsdefinisjon kalles parametre
- Inputs i et funksjonskall kalles argumenter

Eksempel:



Overføring av verdier til funksjoner

Regel 2:

> Antall argumenter i et funksjonskall må være det samme som antall parametre i funksjonen.

Eksempel:

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

verdi = f(0)        # Ikke lovlig
verdi = f(0,1)      # Lovlig
verdi = f(0,1,2)    # Ikke lovlig
```

Overføring av verdier til funksjoner

Regel 3:

> Vi kan gi default-verdier til parametre når vi definerer en funksjon. Da trenger vi ikke å gi verdier til dem i et kall.

Eksempel:

```
def f(x,y,z=2):
   return x+y+z

verdi = f(0)       # Ikke lovlig
verdi = f(0,1)     # Lovlig
verdi = f(0,1,2)   # Lovlig
```

Overføring av verdier til funksjoner

MERK:

Parametre med default-verdier må stå bakerst i listen av parametre!

EKSEMPEL:

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

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

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

Overføring av verdier til funksjoner

Regel 4:

> Vi kan navngi argumenter når vi kaller på en funksjon. Dette kalles *keyword arguments*. Slike argumenter må komme sist i argumentlisten.

Eksempel:

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

verdi = f(x=0,y=1,z=2)   # Lovlig
verdi = f(0,y=1,z=2)     # Lovlig
verdi = f(0,z=2,y=1)     # Lovlig
verdi = f(z=2,0,1)       # Ikke lovlig
```

Overføring av verdier til funksjoner

Regel 5:

> Vi kan bruke *args hvis vi ønsker å tillate et variabelt antall argumenter i et kall.

Eksempel:

```
def f(*args):
  # Nå er args et tuppel
  for k in args:
    print(k)

f()       # Lovlig
f(0)      # Lovlig
f(0,1)    # Lovlig
```

Variabler og parametre

I matematiske funksjoner skiller vi ofte mellom variabler og parametre/koeffisienter.

Eksempler:

$$f(x) = ax^2 + bx + c$$

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

Når vi programmerer slike funksjoner, må vi også ha med parametrene som input-variabler.

Ofte nyttig å gi dem default-verdier.

Eksempel

Anta at vi har en funksjon av t, med parametre A, a og ω:

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

Mulig implementasjon 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)
```

Mange muligheter når vi kaller på funksjonen, f.eks:

```
f(0.5)
f(0.5, A=2)
f(0.5, a=4)
f(0.5, A=3, a=1, omega=4*pi)
```

Quiz 1

Vi har funksjonen

```
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)
```

Quiz 1

Vi har funksjonen

```
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 (y mangler)
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)
```

Quiz 1

Vi har funksjonen

```
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 (y mangler)
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)
```

Quiz 1

Vi har funksjonen

```
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 (y mangler)
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)
```

Quiz 1

Vi har funksjonen

```
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 (y mangler)
r = h(0, 1)                 # Lovlig
r = h(0, 1, 2)              # Lovlig
r = h(x=0, 1, 2)            # IKKE lovlig (navngitt arg først)
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)
```

Quiz 1

Vi har funksjonen

```
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 (y mangler)
r = h(0, 1)                 # Lovlig
r = h(0, 1, 2)              # Lovlig
r = h(x=0, 1, 2)            # IKKE lovlig (navngitt arg først)
r = h(0, y=1)               # Lovlig
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)
```

Quiz 1

Vi har funksjonen

```
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 (y mangler)
r = h(0, 1)                 # Lovlig
r = h(0, 1, 2)              # Lovlig
r = h(x=0, 1, 2)            # IKKE lovlig (navngitt arg først)
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)
```

Quiz 1

Vi har funksjonen

```
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 (y mangler)
r = h(0, 1)                 # Lovlig
r = h(0, 1, 2)              # Lovlig
r = h(x=0, 1, 2)            # IKKE lovlig (navngitt arg først)
r = h(0, y=1)               # Lovlig
r = h(0, 1, z=3)            # Lovlig
r = h(0, 0, x=0)            # IKKE lovlig (x får verdi 2 ganger)
r = h(z=0, x=1)
r = h(z=0, x=1, y=2)
```

Quiz 1

Vi har funksjonen

```
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 (y mangler)
r = h(0, 1)                 # Lovlig
r = h(0, 1, 2)              # Lovlig
r = h(x=0, 1, 2)            # IKKE lovlig (navngitt arg først)
r = h(0, y=1)               # Lovlig
r = h(0, 1, z=3)            # Lovlig
r = h(0, 0, x=0)            # IKKE lovlig (x får verdi 2 ganger)
r = h(z=0, x=1)             # IKKE lovlig (y mangler)
r = h(z=0, x=1, y=2)
```

Quiz 1

Vi har funksjonen

```
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 (y mangler)
r = h(0, 1)             # Lovlig
r = h(0, 1, 2)          # Lovlig
r = h(x=0, 1, 2)        # IKKE lovlig (navngitt arg først)
r = h(0, y=1)           # Lovlig
r = h(0, 1, z=3)        # Lovlig
r = h(0, 0, x=0)        # IKKE lovlig (x får verdi 2 ganger)
r = h(z=0, x=1)         # IKKE lovlig (y mangler)
r = h(z=0, x=1, y=2)    # Lovlig
```

Funksjoner kan ha funksjoner som argumenter

Input-verdier til funksjoner kan selv være funksjoner:

```
def evaluer(g):
    return g(0)

from math import cos
verdi = evaluer(cos)
print(f"Cos(0) = {verdi}")
```

Når vi utfører evaluer(cos) så er det en funksjon og ikke et tall som sendes inn i funksjonen!

Eksempel

Vi kan estimere den annenderiverte til en funksjon f i et gitt punkt x med denne formelen:

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

hvor $h > 0$ er et lite tall (f.eks. h=0.000001).

Python-funksjon som beregner den annenderiverte i et punkt x med formelen over:

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

En liten digresjon

Funksjonen vi nettopp definerte, har standard-verdien h=1E-6. Er det en grunn til å velge akkurat denne verdien?

- Matematisk blir approksimasjonen bedre jo mindre h er.
- Numerisk får vi problemer hvis h blir for liten (fordi maskinen jobber med endelig presisjon)
- I praksis blir det ofte litt prøving og feiling for å finne en god *h*-verdi.

Effekten av å endre h

For å se effekten av å endre h skriver vi et program:

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

def f(t):
  return t**(-6)

for k in range(1,14):
  h = 10**(-k)
  print (f"h = {h:.0e}: {diff2(g,1,h):.5f}")
```

Programmet beregner $f''(1)$ med approksimasjonen vi så på for h=0.1, h=0.01, h=0.001, osv...

Effekten av å endre h

Siden $f(t) = t^{-6}$ så er

$$f'(t) = -6\,t^{-7}$$
$$f''(t) = (-6)(-7)\,t^{-8} = 42\,t^{-8}$$

Dermed er $f''(1) = 42$. Her er svaret fra programmet:

```
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
```

Veldig små h-verdier gir helt feil svar

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

Problem 1: for veldig små h subtraherer vi tall som er nesten like; dette gir 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.

Lambda-funksjoner

For veldig enkle funksjoner kan lambda-funksjoner være et alternativ.

Eksempel:

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

kan defineres på én linje med lambda:

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

Eksempel på bruk:

```
z = diff2(lambda x:x**6, 4)
```

Funksjoner bør dokumenteres!

Vi kan lage en *doc string* som plasseres rett etter funksjons-headeren og i triple anførselstegn.

Eksempel:

```
def line(x0, y0, x1, y1):
    """
    Beregn koeffisientene a og b i uttrykket
    for en rett linje y = a*x + b som passerer
    gjennom (x0,y0) og (x1,y1).
    """
    a = (y1 - y0)/(x1 - x0)
    b = y0 - a*x0
    return a, b
```

assert: en spesiell form for test

Noen ganger ønsker man å stoppe programkjøringen og gi en feilmelding hvis et bestemt krav ikke er oppfylt.

For dette formål har vi assert:

```
assert x > 0, "x må være positiv"
```

Når setningen over utføres, skjer følgende:

Hvis $x > 0$ så skjer ingenting

Hvis $x \leq 0$ så stopper programmet med feilmeldingen "x må være positiv"

Testfunksjoner

Anta at du har skrevet en funksjon som beregner og returnerer et svar.

Hvordan kan vi vite om svaret er det vi ønsker?

Testfunksjoner

Anta at du har skrevet en funksjon som beregner og returnerer et svar.

Hvordan kan vi vite om svaret er det vi ønsker?

Teststrategi:

- Vi lager en *testfunksjon* i Python som kaller på funksjonen med noen utvalgte input-verdier der vi vet hva output skal være.
- Hvis output ikke stemmer med fasiten, gir testfunksjonen en feilmelding.

Krav til testfunksjoner

En testfunksjon skal *ikke* gi utskrift på skjermen hvis funksjonen som testes består testene. Hvis en test feiler, skal det gis feilmelding (AssertionError).

Regler for testfunksjoner:

- Hvis navnet på funksjonen som skal testes er XXX så skal navnet på testfunksjonen være test_XXX
- En testfunksjon skal ikke ha parametre
- En testfunksjon skal ha en assert success, message setning, hvor success:
    - er True hvis testen bestås
    - er False hvis testen ikke bestås

Eksempel 1

```
# Funksjon som skal finne produktet av alle
# verdiene i en liste

def prod(a):
  result = 1
  for e in a:
    result = result * e
  return result

# Vi lager en testfunksjon:

def test_prod():
  a = [3,1,5]              # Eksempel på inputverdi for a
  computed = prod(a)       # Faktisk output fra prod
  expected = 15            # Forventet output fra prod
  success = (computed == expected)
  message = "prod gir feil svar!"
  assert success, message

# Vi kaller på testfunksjonen:

test_prod()
```

Eksempel 2

Vi kan bedre testen ved å legge inn flere test-cases:

```
def prod(a):
    result = 1
    for e in a:
        result = result * e
    return result

def test_prod():
    inputs = [[3,1,5], [2,4,5], [1,0]]
    answers = [15, 40, 0]
    for i in range(len(inputs)):
        computed = prod(inputs[i])
        expected = answers[i]
        success = (computed == expected)
        message = "prod gir feil svar"
        assert success, message

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

Eksempel 3

En svakhet med alle testfunksjonene vi har sett på til nå, er at de tester om computed og expected er *helt identiske*. I praksis kan det være små avrundingsfeil i computed som gjør at computed == expected blir False selv om svaret egentlig er riktig.

For å unngå dette, kan vi i stedet sjekke om avstanden mellom computed og expected er veldig liten.

```
def test_prod():
    inputs = [[3.1, 1.1, 5.2], [2.2, 4.0, 5.9]]
    answers = [17.732, 51.92]
    tolerance = 1e-10
    for inp, expected in zip(inputs, answers):
        computed = prod(inp)
        success = abs(computed-expected) < tolerance
        message = "prod gir feil svar"
        assert success, message
```

Hvordan kjøres testfunksjoner i praksis?

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.pyfor 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_.

Enhetstesting

I praksis må vi teste store programmer for feil på flere ulike måter.

En måte er å sjekke at hver enkelt funksjon i programmet fungerer (droppes i praksis for de aller enkleste funksjonene).

Det kalles enhetstesting.

Programmet må da 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 er hensikten med en testfunksjon.
- 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.