Python for MAT 1110

(revidert versjon våren 2009)

av

Klara Hveberg, Tom Lindstrøm, og Øyvind Ryan

Dette lille notatet gir en kort innføring i Python med tanke på behovene i MAT 1110. Hensikten er å gi deg litt starthjelp slik at du kommer i gang med å bruke språket. Avansert bruk må du lære deg andre steder (se lenkene på kursets hjemmeside). Du kan også få hjelp fra Python s innebygde hjelpemenyer og demo'er (kommandoen `help()` i et Python Shell gir deg en oversikt over hva som er tilgjengelig). Notatet forutsetter at du sitter ved maskinen og taster inn kommandoene vi gjennomgår, men det kan kanskje være lurt å ha bladd fort gjennom på forhånd slik at du vet hva som vil bli tatt opp.

I de første seksjonene i notatet (1-7) konsentrerer vi oss hovedsakelig om hvordan du kan bruke Python til å utføre enkle beregninger. I de siste seksjonene (8-11) ser vi på litt mer avanserte ting som hvordan du tar vare på (utvalgte deler av) arbeidet ditt, hvordan du programmerer i Python , og hvordan du lager dine egne Python -rutiner som du kan kalle på igjen og igjen. Det kan hende at du har bruk for noe av dette stoffet før du er ferdig med de første sju seksjonene. Heftet inneholder også en seksjon om bildebehandling. Denne er ikke pensum i kurset, men kan bli brukt i forbindelse med obligatoriske oppgaver.

Du vil i dette heftet se mange kodeeksempler. I disse har vi brukt farger for å markere forskjellige ting:

- grønt brukes for kommentarer. I eksemplene ser du at en kommentar begynner med `#`. Python tolker det da slik at kommentaren løper til slutten av linjen, slik at man ikke trenger å angi hvor kommentaren slutter. Kommentarene er ikke en del av selve programmet som blir kjørt.
- blått brukes for nøkkelord som `if`, `else`, `def`, etc.
- rødt brukes for å markere tekststrenger

Vi har prøvd å gi mange kommentarer i koden, slik at det skal være mulig å lese seg til hva som skjer. Det er en god vane å legge inn slike forklarende kommentarer i filene. Du vil se at Python -funksjoner inneholder noen kommentarer helt i starten på hvordan input og output for funksjonen skal tolkes. Dette er helt vanlig programmeringsskikk i Python og i mange andre språk.

Det er også laget mye annet stoff ved Universitetet i Oslo for MAT1110 enn det som finnes i dette heftet. Læreboka inneholder for eksempel en del MATLAB-tips, i tillegg til det du kan finne her. Mange morsomme og nyttige MATLAB-oppgaver (både oppvarmingsoppgaver og oppgaver mer for viderekomne) finnes også på `http://www.math.uio.no/~klara/matlab/`. Oppgavene er brukt i MAT1110 de siste årene.

Vi retter en takk til Karl Leikanger og Magnar Bugge for å ha programmert mye av Python-koden i dette heftet.

1 Det aller enkleste

Du åpner Python som et hvilket som helst annet program. I Unix kan du skrive `ipython &`. Du får da opp et "prompt". Vi skal indikere et slikt prompt med >>>, selv om det ikke er akkurat slik `ipython` viser det. I kodeeksemplene i dette heftet vil vi bare vise promptet hvis vi trenger skille mellom våre egne kommandoer, og det Python svarer med. Etter promptet kan du skrive en kommando du vil at Python skal utføre. Skriver du f.eks.

```
3+4
```

og skifter linje, svarer Python med `7`, og gir deg et nytt "prompt" til neste regnestykke (prøv!). Du kan prøve de andre standard regneoperasjonene ved å skrive inn etter tur

```
3-4, 3*4, 3.0/4, 3**4
```

(husk å avslutte med et linjeskift for å utføre kommandoene). Python bruker den vanlige prioriteringsordningen mellom regneoperasjonene slik du er vant til fra avanserte lommeregnere. Derfor gir kommandoen

```
8.0**2.0/3.0
```

noe annet enn

```
8.0**(2.0/3.0)
```

Legg merke til at `2/3` (heltallsdivisjon) og `2.0/3.0` er forskjellige ting i Python.

For å få tilgang til mye brukte matematiske funksjoner i Python bør vi i koden først inkludere linjen

```
from math import *
```

Hvis vi bare er interessert i et par matematiske funksjoner kan vi skrive

```
from math import sin, exp, sqrt, abs
```

Python kjenner alle vanlige (og mange uvanlige!) funksjoner, så skriver du

```
sin(0.94)
```

svarer Python med sinus til 0.94 radianer. Vær oppmerksom på at Python er nøye på multiplikasjonstegn: Skriver du `15sin(0.94)`, svarer Python med en feilmelding: du må skrive `15*sin(0.94)` for å få regnet ut $15 \sin 0.94$.

En funksjon du må passe litt på, er eksponentialfunksjonen e^x som du må skrive som `exp(x)`. Vil du regne ut $e^{2.5}$, skriver du altså `exp(2.5)` (og ikke potenstegnet over). I tillegg til eksponentialfunksjonen heter arcusfunksjonene i Python `arcsin`, `arctan`, osv. I tillegg kan det være greit å vite at kvadratrotfunksjonen heter `sqrt` og at absoluttverdifunksjonen heter `abs`. Skriver du

```
exp(sqrt(abs(-0.7)))
```

regner Python ut $e^{\sqrt{|-0.7|}}$. De andre standardfunksjonene heter det du er vant til: `sin, cos, tan, log`.

Vår gamle venn π kalles i Python for `pi`. Kommandoen

```
pi
```

gir svaret `3.1415926535897931`.
Ber du Python regne ut $\sin(\pi)$, får du deg kanskje en overraskelse. Istedenfor 0, gir Python deg svaret `1.2246e-16`. Dette skyldes at Python primært regner med avrundede tall.

Python har også funksjoner som runder av tall på flere måter:

- `floor` runder ned til nærmeste hele tall,
- `ceil` runder opp til nærmeste hele tall,
- `round` runder av til nærmeste hele tall.

Skal du bruke et tall flere ganger i en beregning, kan det være greit å gi det et navn. Ønsker du at 0.3124 heretter skal betegnes med `a`, skriver du rett og slett

```
a=0.3124
```

Navnsetter du på tilsvarende måte

```
b=2.41
```

kan du regne ut produktet ved å skrive

```
a*b
```

La oss avslutte denne innledende seksjonen med noen Python -knep det kan være greit å vite om (blir det for mye på en gang, får du heller komme tilbake til denne delen senere). I Python kan du ikke endre på en kommando som er blitt eksekvert. Har du laget en liten trykkfeil, må du derfor føre inn kommandoen på nytt. Dette kan du gjøre ved klipping-og-liming, men du kan også bruke piltastene opp og ned.

Får du problemer med en kommando og ønsker å avbryte, trykker du Ctrl-c (altså kontrolltasten og c samtidig).

0ppgaver til Seksjon 1

1. Bruk Python til å regne ut: $2.2 + 4.7$, $5/7$, 3^2, $\frac{2\cdot 3-4^2}{13-2.2^2}$

2. La Python regne ut verdiene, og sjekk at resultatene er rimelige: e^1, $\sqrt{16}$, $\cos\pi$, $\sin\frac{\pi}{6}$, $\tan\frac{\pi}{4}$, $\arcsin\frac{1}{2}$, $\arctan 1$.

3. Definer $x = 0.762$ og $y = \sqrt{9.56} + e^{-1}$ og regn ut: $x + y$, xy, $\frac{x}{y}$ og $\sin x^2 y$

2 Matriser og vektorer

Støtten for matriser i Python finner vi i pakken `numpy`, slik at vi må skrive

```
from numpy import *
```

for å få tilgang til funksjonalitet for matriser (alternativt kan vi importere bare de klassene fra `numpy` vi trenger). I en del korte kodeeksempler er denne linjen droppet i det følgende. De viktigste klassene vi trenger fra `numpy` er `matrix` og `array`. Vi skal først se på `matrix`, og komme tilbake til `array` senere. Dersom vi ønsker å definere matrisen

$$A = \begin{pmatrix} 2 & -3 & 1 \\ 0 & 1 & -3 \\ -4 & 2 & 1 \end{pmatrix}$$

i Python , skriver vi

```
from numpy import *

A=matrix([[2,-3,1],
[0,1,3],
[-4,2,1]])
```

(legg merke til at vi bruker hakeparenteser for å definere matriser). Synes du dette tar for stor plass, kan du isteden skrive

```
A=matrix([[2,-3,1],[0,1,3],[-4,2,1]])
```

Python utfører regneoperasjoner for matriser med enkle kommandoer. Har du lagt inn matrisene A og B, vil kommandoene

```
A+B
```

```
A-B
```

```
A*B
```

få Python til å regne ut henholdsvis summen $A + B$, differensen $A - B$ og produktet AB. Dette forutsetter at matrisene har riktige dimensjoner slik at regneoperasjonene er definert (hvis ikke svarer Python med en feilmelding av typen: `Objects are not aligned`). Kommandoen

```
3*A
```

ganger matrisen A med tallet 3, og kommandoen

```
A**7
```

regner ut sjuende potensen til A (dvs. A ganget med seg selv 7 ganger). For å finne den transponerte matrisen A^T skriver du

```
A.T
```

Du finner flere nyttige funksjoner i pakken `numpy.linalg`. For å finne den inverse matrisen A^{-1} kan du for eksempel skrive

```
linalg.inv(A)
```

Python regner ut determinanten til A når du skriver

```
linalg.det(A)
```

Det er også lett å finne egenverdier og egenvektorer. Skriver du

```
D,V = linalg.eig(A)
```

vil Python definere to matriser V og D. Søylene i matrisen V er egenvektorene til A, mens D er et array med egenverdiene til A. Egenvektorene og egenverdiene kommer i samme rekkefølge slik at den første egenverdien tilhører den første egenvektoren osv. Her er et eksempel på en kjøring:

Listing 1: Utregning av egenverdier og egenvektorer

```
>>> from numpy import *
>>> A = matrix([[3,1,2],     # taster inn matrisen A
                [3,0,4],
                [2,1,4]])
>>> u,v = linalg.eig(A)      # ber Python finne egenvektorer og -verdier
>>> print u                  # ber Python skrive ut u
[ 6.6208359   1.43309508 -1.05393098]
>>> print v                  # ber Python skrive ut v
[[-0.50695285 -0.79426125  0.18076968]
 [-0.60226696  0.03095748 -0.97598223]
 [-0.61666305  0.60678719  0.12157722]]
```

Dette forteller oss at A har egenvektorer

$$\begin{pmatrix} -0.5070 \\ -0.6023 \\ -0.6167 \end{pmatrix}, \quad \begin{pmatrix} -0.7943 \\ 0.0310 \\ 0.6068 \end{pmatrix} \quad \text{og} \quad \begin{pmatrix} 0.1808 \\ -0.9760 \\ 0.1216 \end{pmatrix}$$

med egenverdier henholdsvis 6.6208, 1.4331 og -1.0539. Vær oppmerksom på at Python alltid normaliserer egenvektorene slik at de har lengde én, og at dette kan få dem til å se mer kompliserte ut enn nødvendig. Husk også at de tre siste kommandoene vi har sett på (`inv, det, eig`) forutsetter at A er en kvadratisk matrise.

Python oppfatter vektorer som spesialtilfeller av matriser. En radvektor $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ er altså en $1 \times n$-matrise. Vektoren $\mathbf{b} = (-2, 5, 6, -4, 0)$ lastes derfor inn med kommandoen

```
b=matrix([[-2,5,6,-4,0]])
```

En søylevektor

$$\mathbf{c} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix}$$

oppfattes som en $m \times 1$-matrise. Du kan laste inn vektoren $\mathbf{c} = \begin{pmatrix} 7 \\ -3 \\ 2 \end{pmatrix}$ ved for eksempel å skrive

```
c=matrix([[7],[-3],[2]])
```

Når du skriver vektorer i Python er det viktig å ha tenkt igjennom om du ønsker radvektorer eller søylevektorer; du kan ikke regne med at Python skjønner at du mener en søylevektor når du skriver en radvektor!

Har du lastet inn en matrise A og en søylevektor $\mathbf{c}$ kan du få Python til å regne ut produktet $A\mathbf{c}$ ved å skrive

```
A*c
```

Python har spesialkommandoer for å regne ut skalar- og vektorprodukt av vektorer. Disse funksjonene forutsetter at objektene er av typen `array` i `numpy`

```
dot(a,b)
cross(a,b)
```

Den siste kommandoen forutsetter naturlig nok at vektorene er tredimensjonale.

Oppgaver til Seksjon 2

1. La

$$A = \begin{pmatrix} 1 & 3 & 4 & 6 \\ -1 & 1 & 3 & -5 \\ 2 & -3 & 1 & 6 \\ 2 & 3 & -2 & 1 \end{pmatrix} \text{ og } B = \begin{pmatrix} 2 & 2 & -1 & 4 \\ 2 & -1 & 4 & 6 \\ 2 & 3 & 2 & -1 \\ -1 & 4 & -2 & 5 \end{pmatrix}.$$

Bruk Python til å regne ut; A^T, B^T, $(AB)^T$, A^TB^T, B^TA^T, A^{-1}, B^{-1}, $(AB)^{-1}$, $A^{-1}B^{-1}$, $B^{-1}A^{-1}$. Blir noen av resultatene like?

2. Vi har to vektorer $a = (1, -2, 3)$ og $b = (2, 2, -4)$. Sjekk at lengdene til vektorene kan finnes ved kommandoene `linalg.norm(a)` og `linalg.norm(b)`. Forklar at du kan finne vinkelen mellom vektorene ved kommandoen

```
acos(dot(a,b)/(linalg.norm(a)*linalg.norm(b)))
```

Finn vinkelen.

3. La

$$A = \begin{pmatrix} 1 & 2 & -1 \\ 3 & -1 & 0 \\ -4 & 0 & 2 \end{pmatrix}$$

Finn A^{-1}, A^T og $\det(A)$. Finn også egenverdiene og egenvektorene til A.

3 Komponentvise operasjoner

Ovenfor har vi sett på de algebraiske operasjonene som brukes mest i vanlig matriseregning. Det finnes imidlertid andre operasjoner slik som det komponentvise matriseproduktet (eller *Hadamard-produktet*) der produktet av matrisene

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \text{og} \quad \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix}$$

er

$$\begin{pmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \cdots & a_{2n}b_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \cdots & a_{mn}b_{mn} \end{pmatrix}$$

Selv om slike komponentvise operasjoner ikke brukes mye i lineær algebra, er de viktige i Python siden Python bruker matriser til mer enn tradisjonell matrise-regning. Python støtter slike operasjoner ved hjelp av klassen `array`. der alle operasjoner som var tillatt for klassen `matrix` også eksisterer, men er i stedet komponentvise. Et objekt av typen `matrix` kan konverteres til et `array` ved hjelp av metoden `asarray` i `numpy`. Skriver du f.eks.

```
A = asarray(A)
B = asarray(B)
A*B     # Komponentvis multiplikasjon
A/B     # Komponentvis divisjon
A**B    # Komponentvis potens
```

vil Python regne ut matrisene der den ij-te komponenten er $a_{ij}b_{ij}$, $\frac{a_{ij}}{b_{ij}}$, og $a_{ij}^{b_{ij}}$, respektive. Vi kan også bruke disse operasjonene når den ene matrisen erstattes med et tall; f.eks. vil

```
3.0/A   # Komponentvis divisjon av 3
A**2    # Komponentvis kvadrat
```

produsere matrisene med komponenter $\frac{3}{a_{ij}}$ og ${a_{ij}}^2$, respektive. En del kjente funksjoner er også tilgjengelige på matriser, er allerede komponentvise, og kan kombineres: Skriver vi

```
sin(A)       # Komponentvis sinus
exp(A)       # Komponentvis eksponentialfunksjon
exp(A*B**2)  # Kombinasjon av komponentvise operasjoner
```

får vi matrisene med komponenter $\sin a_{ij}$, $\exp a_{ij}$, og $e^{a_{ij}b_{ij}^2}$, respektive. En annen viktig komponentvis funksjon er random-generering av tall:

```
from numpy import random

random.rand(m,n)   # Tilfeldig generert mxn-matrise
random.rand(n)     # Tilfeldig generert vektor av lengde n
random.rand()      # Tilfeldig generert tall
```

Alle tall blir her tilfeldig generert mellom 0 og 1. Når vi skal illustrere at en setning holder kommer vi ofte til å lage en tilfeldig generert matrise, og vise at setningen holder for denne matrisen. Som et eksempel, studer følgende kode:

Listing 2: Bruk av `rand` til å illustrere kjente setninger

```
A = matrix(random.rand(4,4))
B = matrix(random.rand(4,4))

print (A+B).T - A.T - B.T
print (A*B).T - B.T * A.T
```

Ser du hvilke to kjente setninger som blir illustrert her hvis du kjører koden?

Som vi skal se i neste seksjon, er de komponentvise operasjonene spesielt nyttige når vi skal tegne grafer, men det finnes også noen enklere anvendelser som vi kan se på allerede nå. Python har egne kommandoer for å finne summer og produkter. Har du lagt inn en vektor c, kan du finne summen og produktet til komponentene i c ved å skrive

```
sum(c)
prod(c)
```

Skal vi finne summen av de 10 første naturlige tallene, kan vi dermed skrive

```
a=array([1,2,3,4,5,6,7,8,9,10])
print sum(a)
```

Hvis vi også vil ha summen av de 10 første kvadrattallene, kan vi skrive

```
sum(a**2)
```

Dette er vel og bra så lenge vi har korte summer, men hva hvis vi ønsker summen av de hundre første kvadrattallene? Det blir ganske kjedelig å taste inn vektoren $(1, 2, 3, \ldots, 100)$ for hånd. Python har løsninger for dette. Taster du

```
a=range(1,101)
```

definerer du a til å være vektoren $(1, 2, 3, \ldots, 100)$. Skriver du isteden

```
a=range(1,101,2)
```

blir a vektoren bestående av alle oddetallene mindre enn 100. Generelt vil

```
a=range(b,c,h)
```

definere a til å være vektoren med b som førstekomponent, $b + h$ som annenkomponent, $b + 2h$ som tredjekomponent osv. inntil vi kommer til c (den siste komponenten vil altså være det største tallet på formen $a + nh$ som er mindre enn c). Skal vi regne ut summen av kvadratene av alle partall opptil 100, kan vi altså skrive

```
a=range(2,101,2)
sum(array(a)**2)
```

Til slutt nevner vi kommandoen `linspace` som ofte er nyttig hvis intervallet vi skal dele opp har "uregelmessige" endepunkter. Skriver vi f.eks.

```
a=linspace(0,pi,50);
```

får vi definert `a` som en vektor med 50 komponenter der første komponent er 0, siste komponent er π og avstanden mellom én komponent og den neste allltid er den samme.

0ppgaver til Seksjon 3

1. Legg inn disse n-tuplene i Python : $(1, -9, 7, 5, -7)$, $(\pi, -14, e, 7/3)$, $(1, 3, 5, 7, 9, \ldots, 99)$, $(124, 120, 116, 112 \ldots, 4, 0)$.

2. Legg inn tuppelet $(1, 2, 4, 8, 16, \ldots 4096)$ i Python og finn summen.

3. Legg inn $(0, \frac{1}{100}, \frac{2}{100}, \ldots, 1)$ i Python . Bruk denne partisjonen til å lage en nedre og øvre trappesum for funksjonen $f(x) = x^2$ over intervallet $[0, 1]$ (husk MAT1100!), og regn ut disse summene. Sammenlign med integralet $\int_0^1 x^2\, dx$.

4. Bruk Python til å regne ut produktet $\frac{1}{2} \cdot \frac{3}{4} \cdots \frac{99}{100}$.

4 Grafer

For å plotte i Python trenger vi å importene pakken `scitools.easyviz`, ved for eksempel å skrive

```
from scitools.easyviz import *
```

Anta at $x = (x_1, x_2, \ldots, x_n)$ og $y = (y_1, y_2, \ldots, y_n)$ er to n-tupler. Kommandoen

```
plot(x,y)
```

lager en strektegning der punktet (x_1, y_1) er forbundet til (x_2, y_2) med en strek, (x_2, y_2) er forbundet til (x_3, y_3) med en strek osv. Dersom du istedet skriver

```
plot(x)
```

(altså én vektor istedenfor to), lager Python en strektegning mellom punktene $(1, x_1), (2, x_2), (3, x_3)$ osv. Vær oppmerksom på at figuren kommer i et eget vindu, og at dette vinduet kan gjemme seg bak andre vinduer!

Vi kan utnytte metoden ovenfor til å plotte grafer av funksjoner. Trikset er å velge punktene som skal forbindes til å være (tettliggende) punkter på funksjonsgrafen. La oss se hvordan vi kan bruke metoden til å tegne grafen til $\sin \frac{1}{x}$ over intervallet $[-\pi, \pi]$. Vi velger først et tuppel som gir oppdeling av intervallet i mange små biter:

```
x=linspace(-pi,pi,100)
```

Så velger vi tuppelet av tilhørende funksjonsverdier:

```
y=sin(1.0/x)
```

Til slutt plotter vi punktene

```
plot(x,y)
```

Hva hvis vi har lyst til å tegne en graf til i samme vindu? Da gir vi først kommandoen

```
hold('on')
```

og taster så inn en ny funksjon. Hvis den nye funksjonen er $z = x^2 \sin \frac{1}{x}$, skriver vi først

```
z=x**2*sin(1.0/x)
```

og så plotter vi punktene:

```
plot(x,z)
```

Grafene du lager vil fortsette å komme i samme vindu inntil du gir kommandoen

```
hold('off')
```

Det hender at du vil lage en ny figur i et nytt vindu og samtidig beholde den gamle i det gamle vinduet. Du kan aktivere et eksisterende (eller nytt) vindu med kommandoen

```
figure(2)
```

Etter denne kommandoen vil alle plott havne i vindu 2.

Vi skal se på noen tilleggskommandoer som kan være nyttige å kunne. Dersom du selv vil velge størrelsen på vinduet grafene skal vises i, kan du bruke en kommando av typen

```
axis([-pi, pi, -2.5, 2.5])
```

Vil du ha et kvadratisk vindu, skriver du

```
axis('square')
```

mens

```
axis('equal')
```

gir deg samme målestokk på begge aksene. Du kan gi figuren en tittel ved å skrive noe slikt som

```
title('Grafen til en vakker funksjon')
```

og du kan sette navn på aksene ved å skrive

```
xlabel('x-akse')
ylabel('y-akse')
```

Det er lett å angi farge på grafene, samt måte grafen skal tegnes:

```
plot(x,y,'r')          # en rød graf
plot(x,y,'g')          # gir en grønn graf
plot(x,y,'r--')        # gir en rød, stiplet graf
plot(x,y,'g:')         # gir en grønn, prikket graf
```

Av og til ønsker man å ha flere grafer side om side i den samme figuren. Kommandoen

```
subplot(m,n,p)
```

deler vinduet i $m \times n$-delvinduer, og sørger for at den neste plot-kommandoen blir utført i det p-te delvinduet.

Hvis du ønsker å ta vare på en figur for å kunne bruke den i et annet dokument senere, kan det være lurt å lagre den som en eps-fil (encapsulated postscript format). Du kan skrive innholdet i figurvinduet til filen figur.eps ved kommandoen

```
hardcopy('figur.eps')
```

eller eventuelt

```
hardcopy('figur.eps',color=True)
```

for å få figuren i farger. Du kan få andre formater som png ved å modifisere kommandoen på denne måten

```
hardcopy('figur.png')
```

0ppgaver til Seksjon 4

1. Legg inn 6-tuplene $a = (3, 1, -2, 5, 4, 3)$ og $b = (4, 1, -1, 5, 3, 1)$ i Python og utfør kommandoen `plot(a,b)`. Utfør også kommandoene `plot(a)` og `plot(b)`, og bruk `hold on` til å sørge for at de to siste figurene kommer i samme vindu.

2. Bruk kommandoen `plot` til å lage en enkel strektegning av et hus.

3. Bruk Python til å tegne grafen til $f(x) = x^3 - 1$ over intervallet $[-1, 1]$. Legg så inn grafen til $g(x) = 3x^2$ i samme koordinatsystem, og velg forskjellig farge på de to grafene.

4. Bruk Python til å tegne grafen til funksjonen $f(x) = \sin \frac{1}{x}$ over intervallet $[-1, 1]$. Bruk først skrittlengde $\frac{1}{100}$ langs x-aksen. Tegn grafen på nytt med skrittlengde $\frac{1}{10000}$.

5 Tredimensjonale grafer

Python har flere kommandoer som kan brukes til å lage tredimensjonale figurer. Vi skal se på et par av de enkleste. Grafen til en funksjon $z = f(x, y)$ tegnes ved å plotte funksjonsverdiene over et nett av gitterpunkter i xy-planet.

Kommandoen `mesh` lager konturer av grafen ved å forbinde plottepunktene på grafen med rette linjestykker. Resultatet ser ut som et fiskegarn med knuter i plottepunktene. Varianten `meshc` tegner i tillegg nivåkurver til funksjonen i xy-planet. La oss se hvordan vi kan tegne grafen til funksjonen $z = f(x, y) = xy \sin(xy)$ over området $-4 \leq x \leq 4, -2 \leq y \leq 2$.

Vi lager først en oppdeling av de to intervallene vi er interessert i. I koden vår har vi valgt å dele opp begge intervallene i skritt med lengde 0.05, men du kan godt velge en finere eller grovere oppdeling. Neste skritt er å lage et rutenett av oppdelingene våre, regne ut funksjonsverdiene, og til slutt plotte:

Listing 3: Plotting i tre dimensjoner

```
from math import *
```

```
from numpy import *
from scitools.easyviz import *

r=arange(-4,4,0.05,float)       # Lag en oppdeling av
s=arange(-2,2,0.05,float)       # intervallene vi er interessert i
x,y = meshgrid(r,s,sparse=False,indexing='ij') # Lag et rutenett
                                               # av oppdelingene våre
z=x*y*sin(x*y)                  # Regn ut funksjonsverdiene
mesh(x,y,z)                     # selve plottingen
```

Grafen kommer opp i et eget figurvindu akkurat som for todimensjonale figurer. Velger du `surf(x,y,z)` istedenfor `mesh(x,y,z)`, får du en graf der flateelementene er fargelagt. Ved å holde musknappen inne kan dreie flaten i rommet. Dette er ofte nødvendig for å få et godt inntrykk av hvordan grafen ser ut! Ønsker du bare å få ut nivåkurvene til en flate, kan du bruke kommandoen

```
contour(x,y,z)
```

Du kan regulere antall nivåkurver ved å skrive

```
contour(x,y,z,n)
```

der n er antall nivåkurver du ønsker.

Kommandoen `plot3` er nyttig når du skal plotte parametriserte kurver i tre dimensjoner. Skriver du

```
t=linspace(0,10*pi,100)
x=sin(t)
y=cos(t)
z=t
plot3(x,y,z)
```

tegner Python kurven $\mathbf{r}(t) = (\sin t, \cos t, t)$ for $t \in [0, 10\pi]$.

Python inneholder også støtte for å tegne vektorfelt. Skriver du

```
quiver(x,y,u,v)
```

vil et vektorfelt bli tegnet opp der `x`- og `y`-vektorene spesifiserer punktene der vektorfeltet skal tegnes opp, og der `u`- og `v`-vektorene er vektorfeltets verdi i disse punktene. Grafisk betyr dette at det i ethvert punkt (x, y) blir tegnet opp en vektor (u, v). Det kan være lurt å ikke bruke for mange punkter (x, y), siden det da vil tegnes så mange vektorer at disse vil kollidere med hverandre.

0ppgaver til Seksjon 5

1. Tegn grafene til disse funksjonene med Python : $f(x, y) = x^2y^2$, $g(x, y) = \frac{\sin x}{y^2} + x^2$, $h(x, y) = \sin(e^{x+y})$. Vri på flatene for å få et best mulig inntrykk.

2. Bruk Python til å tegne kurvene $\mathbf{r}_1(t) = (t, t^2, \sin t)$ og $\mathbf{r}_2(t) = (\sin^2 t, \cos^2 t, e^{-t})$. Vri på koordinatsystemet for å se kurvene best mulig.

3. Bruk kommandoen `plot3` til å lage en tredimensjonal strektegning av en terning.

6 Mer om matriser

Siden matriser står så sentralt i Python , kan det være greit å vite hvordan man kan manipulere matriser på en effektiv måte. Spesielt nyttig er det å kunne legge sammen elementer i en matrise på forskjellige måter. Dette kan vi gjøre med funksjonen `sum` på følgende måter (`A` er en matrise):

Listing 4: Summering av elementer i matriser på flere måter

```
A.sum(0)      # Returnerer en radvektor der hvert element er
              # summen av tilsvarende søyle
A.sum(1)      # Returnerer en søylevektor der hvert element er
              # summen av tilsvarende rad
A.sum()       # Returnerer summen av alle elementene i matrisen
```

Dette kan brukes for eksempel til å regne ut middelverdien i en matrise, og det vi kaller for *Frobenius normen* til en matrise (ikke pensum):

Listing 5: Anvendelser av `sum`-funksjonen.

```
m,n = shape(A)        # m og n er antall rader og søyler i matrisen
A.sum()/(m*n)         # Regner ut middelverdien i matrisen
sqrt(sum(A**2))       # regner ut Frobenius normen til en matrise
                      # (ikke pensum)
```

Mens `sum` lager en vektor/skalar fra en matrise, så finnes det også Python -kommandoer som går den motsatte veien. Et eksempel er kommandoen `diag`, som lager en diagonalmatrise med elementene i input-vektoren på diagonalen.

Python gir deg blant annet mange muligheter til å sette sammen og ta fra hverandre matriser. Dersom A er 3×3-matrisen

$$A = \begin{pmatrix} 2 & -3 & 1 \\ 0 & 1 & -3 \\ -4 & 2 & 1 \end{pmatrix}$$

og B er 3×2-matrisen

$$B = \begin{pmatrix} 7 & 4 \\ 2 & 5 \\ -1 & 3 \end{pmatrix}$$

kan vi skjøte sammen A og B til 3×5-matrisen

$$C = \begin{pmatrix} 2 & -3 & 1 & 7 & 4 \\ 0 & 1 & -3 & 2 & 5 \\ -4 & 2 & 1 & -1 & 3 \end{pmatrix}$$

ved å gi kommandoen

```
C=hstack((A,B))
```

Du kan skrive ut komponentene til en matrise med enkle kommandoer:

```
A[0,1]
```

skriver ut elementet i første rad, andre søyle i A . Vil du skrive ut hele den første raden, gir du kommandoen

```
A[0,:]
```

mens

```
A[:,1]
```

skriver ut den andre søylen. Du kan også bruke kolon-notasjon til å plukke ut andre deler av en matrise. Starter vi med 3×5-matrisen C ovenfor, vil

```
C[1:3,0:4]
```

gi deg undermatrisen

$$\begin{pmatrix} 0 & 1 & -3 & 2 \\ -4 & 2 & 1 & -1 \end{pmatrix}$$

som består av komponentene som ligger fra annen til tredje rad (det er dette `1:3` står for) og fra første til fjerde søyle (det er dette `0:4` står for). Kommandoen

```
hstack((C[[0,2],1],C[[0,2],4]))
```

gir deg matrisen

$$\begin{pmatrix} -3 & 4 \\ 2 & 3 \end{pmatrix}$$

bestående av elementene som ligger i første og tredje rad og annen og femte søyle. Du kan også bruke denne notasjonen til å bytte om på radene eller søylene til en matrise. Skriver du

```
C[[2,0,1],:]
```

får du ut matrisen

$$\begin{pmatrix} -4 & 2 & 1 & -1 & 3 \\ 2 & -3 & 1 & 7 & 4 \\ 0 & 1 & -3 & 2 & 5 \end{pmatrix}$$

der radene i den opprinnelige matrisen C nå kommer i rekkefølgen $3, 1, 2$. Kommandoen

```
C[:,[2,0,1,4,3]]
```

vil på tilsvarende måte bytte om på søylene i C.

Som tidligere nevnt, kan du *transponere* matrisen C (dvs. bytte om rader og søyler) ved å skrive

```
C.T
```

Resultatet er

$$C^T = \begin{pmatrix} 2 & 0 & -4 \\ -3 & 1 & 2 \\ 1 & -3 & 1 \\ 7 & 2 & -1 \\ 4 & 5 & 3 \end{pmatrix}$$

Legg merke til at hvis

$$a = (a_1, a_2, \ldots, a_n)$$

er en radvektor, vil den transponerte være søylevektoren

$$a^T = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

Det finnes mange andre spesialkommandoer for å manipulere matriser i Python . Her følger noen kommandoer man ofte kan ha nytte av. Noen av de første har vi mer eller mindre vært borti tidligere, men vi gjentar dem her for å ha alt samlet på et sted:

```
x=linspace(2,4,12) # 12-tuppelet med 12 tall jevnt fordelt fra 2 til 4
x=range(1,11)      # 10-tuppelet med alle heltallene fra 1 til 10
x=range(1,11,2)    # 5-tuppelet med annethvert tall fra 1 til 10
                   # 2 angir steglengden
A=zeros((3,4))     # 3x4-matrisen med bare nuller
A=ones((3,4))      # 3x4-matrisen med bare enere
A=eye(3,4)         # 3x4-matrisen med enere på hoveddiagonalen,
                   # nuller ellers
A=random.rand(3,4) # 3x4-matrise med tilfeldige tall mellom 0 og 1
```

Som et enkelt eksempel tar vi med at

```
A=eye(3,4)
```

gir matrisen

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

0ppgaver til Seksjon 6

1. Skriv inn matrisene

$$A = \begin{pmatrix} 2 & -3 & 1 \\ 4 & 1 & -5 \\ 1 & 2 & -1 \end{pmatrix}$$

og

$$B = \begin{pmatrix} -1 & 3 & 2 \\ 4 & 5 & 1 \\ 0 & 2 & -1 \end{pmatrix}$$

i Python og gjennomfør operasjonene

```
C=hstack((A,B))
C[1,3]
C[:,[1,2]]
C[[0,2],2:5]
```

2. Bruk matrisen C fra forrige oppgave. Undersøk hva som skjer med matrisen når du skriver

```
C[:,2]=2*C[:,2]
C[[0,3],:]=4*C[[0,3],:]
```

3. Undersøk hva kommandoen

```
vstack((A,B))
```

gjør når A og B er to matriser.

4. Bruk kommandoene `random.rand(2,2)`, `random.rand(3,3)`, `random.rand(4,4)`, og `random.rand(5,5)` til å generere "tilfeldige" matriser. Finn egenverdiene og egenvektorene i hvert enkelt tilfelle. Hvor mange forskjellige egenverdier ser det ut til at en typisk $n \times n$-matrise har?

7 Radoperasjoner i Python

Denne seksjonen forutsetter at du kjenner til radoperasjoner for matriser, og at du kan bringe en matrise til (redusert) trappeform. Har du ikke lært dette ennå, kan du trygt hoppe over denne seksjonen så lenge.

Python har en egen kommando som bringer en matrise på redusert trappeform. Den heter `rref` (for *reduced row echelon form*), og finnes i modulen `MAT1120lib`, som altså må importeres eksplisitt. Denne modulen laget vi ved Universitetet i Oslo, siden klassen `linalg` i `numpy` ikke inneholder alle funksjoner som trengs innen lineær algebra, slik som `rref`. Andre slike funksjoner som `MAT1120lib` inneholder er `null` (som finner en basis for nullrommet til en matrise), og `orth` (som finner en basis for søylerommet til en matrise). Nedenfor viser vi hvordan `rref` er implementert i `MAT1120lib`.

Listing 6: `rref`-implementasjon med Python

```
def rref(M,*args):
    """
    rref(M,toleranse)
    M: 2-D matrise. Største neglisjerbare element.
    Bestemt av algoritme i programmet dersom ingen ting spesifiseres.

    Implementering av Gauss-Jordan algoritmen.
    Bringer matrise M (2-D array) på redusert trappeform.

    Bruker lignende algoritme som i matlabs rref funksjon.
    Fjerner små pivotelementer (<tol) som kan skyldes avrundingsfeil.
    For penere utskrifter settes alle elementer < tol lik 0
    i arrayen som returneres.

    OBS: Denne algoritmen kan man ikke stole blindt på.
    Kan for mange ''nesten singulære'' matriser gi helt feil svar.
    """
    # Vi forsikrer oss om at M er et array-objekt med float elementer.
    M=matrix(M,float)
```

```
    if rank(M)!=2:
        print 'Feil: Input må være et todimensjonalt objekt.'
        return

    (m,n) = M.shape
    # Leser inn minste neglisjerbare element.
    if len(args)<2:
        # Finner største avrundingsfeil
        maxabs = max(max(asarray(abs(M).sum(0))))
        toleranse = 2.22044*10**(-16) * max([m,n]) * maxabs
    else:
        toleranse=args[1]

    soylenr=-1 # holder orden på hvilken pivot søyle
    for y in xrange(m):
        soylenr += 1
        while soylenr<n:
            # Finn raden blant de gjenværende radene,
            # der 1. element har størst absoluttverdi.
            Mabsolute=abs(M[y:,soylenr])
            storsteelement=Mabsolute.argmax()+y

            # Sjekk om største element er mindre enn toleransen.
            # I så fall: slett raden.
            # Ellers: bruk raden som pivot-rad, og avbryt loopen.
            if Mabsolute[storsteelement-y,0]<toleranse:
                M[storsteelement:,soylenr]=0
                soylenr+=1
            else:
                # bytt om rader
                M[[y,storsteelement],:] = M[[storsteelement,y],:]
                break

        if soylenr<n:
            # Normaliser pivotrad
            M[y,soylenr:]/=M[y,soylenr]

            # Fjern de resterende elementene i søylen
            for z in xrange(m):
                if z!=y:
                    M[z,soylenr:] -= M[y,soylenr:] *  M[z,soylenr]

    # Setter alle tall mindre enn toleranse lik 0
    M_bool=abs(M)>toleranse
    M =matrix(asarray(M)*asarray(M_bool)) # Komponentvis produkt

    return M
```

`MAT1120lib` er brukt i flere av eksemplene senere i heftet . Starter vi med ma-

trisen

$$A = \begin{pmatrix} 1 & 4 & -5 & 7 & 3 & 2 & 8 & -1 \\ 2 & 3 & -2 & 5 & 3 & 3 & 7 & 8 \\ -1 & 1 & 2 & 3 & -1 & 4 & 4 & 4 \end{pmatrix}$$

leder kommandoen

```
B=rref(A)
```

til den reduserte trappeformen

$$B = \begin{pmatrix} 1 & 0 & 0 & -0.48 & 0.88 & -0.2 & -0.04 & 3.36 \\ 0 & 1 & 0 & 2.12 & 0.28 & 1.8 & 2.76 & 2.16 \\ 0 & 0 & 1 & 0.2 & -0.2 & 1 & 0.6 & 2.6 \end{pmatrix}$$

Du kan også bruke Python til å foreta nøyaktig de radoperasjonene du vil. Lar du

$$C = \begin{pmatrix} 2 & -3 & 1 & 7 & 4 \\ 0 & 1 & -3 & 2 & 5 \\ -4 & 2 & 1 & -1 & 3 \end{pmatrix}$$

være matrisen fra forrige seksjon, har vi allerede sett hvordan vi kan bytte om på to rader ved å bruke en kommando av typen

```
C[[3,2,1],:]
```

Denne kommandoen gir deg en ny matrise der rad 1 og 3 er ombyttet. Taster du

```
C[0,:]=2*C[0,:]
```

svarer Python med

```
matrix([[ 4, -6,  2, 14,  8],
        [ 0,  1, -3,  2,  5],
        [-4,  2,  1, -1,  3]])
```

Den har altså ganget den første raden med 2. Skriver du så

```
C[3,:]=C[3,:]+C[1,;]
```

legger Python den første raden til den siste, og du får

```
matrix([[4,-6, 2,14,8],
        [0,1,-3,2,5],
        [0,-4,3,13,11]])
```

Du kunne ha slått sammen disse operasjonene til én ved å skrive

```
C[3,:]=C[3,:]+2*C[1,:]
```

(dersom du prøver den siste kommandoen, må du huske å tilbakeføre C til den opprinnelige verdien først!)

Man kan lure på hva som er vitsen med å kunne foreta radoperasjoner "for hånd" på denne måten når Python har kommandoen `rref` innebygget. Når man

bruker Python som et verktøy, støter man imidlertid ofte på matriser med spesiell struktur, f.eks. svært mange nuller. Da er det ofte mer effektivt selv å programmere hvilke radoperasjoner Python skal gjøre enn å kjøre en standardkommando som `rref` som ikke tar hensyn til strukturen til matrisene.

Radoperasjoner i Python kan brukes til å løse lineære likningssystemer, men du har også muligheten til å finne løsningen med én kommando. Dersom A er en ikke-singulær, kvadratisk matrise og b er en vektor, kan du løse vektorlikningen $Ax = b$ ved kommandoen

```
linalg.solve(A,b)
```

Velger vi for eksempel

$$A = \begin{pmatrix} 1 & -1 & 4 & 3 \\ 2 & 1 & -4 & 5 \\ 6 & 3 & 1 & -2 \\ 3 & 3 & -2 & 4 \end{pmatrix}$$

og

$$b = \begin{pmatrix} 1 \\ 3 \\ 0 \\ -2 \end{pmatrix}$$

gir kommandoen

```
linalg.solve(A,b)
```

svaret

$$x = \begin{pmatrix} 1.3537 \\ -2.4784 \\ -0.7023 \\ -0.0076 \end{pmatrix}$$

0ppgaver til Seksjon 7

1. Skriv inn matrisene

$$A = \begin{pmatrix} 2 & -3 & 1 & 1 & 4 \\ 4 & 1 & -5 & 2 & -1 \\ 1 & 2 & -1 & 2 & -1 \end{pmatrix}$$

og

$$B = \begin{pmatrix} -1 & 3 & 2 & 3 & 1 \\ 4 & 5 & 1 & 4 & 4 \\ 0 & 2 & -1 & -3 & -1 \end{pmatrix}$$

i Python og gjennomfør operasjonene `rref(A)` og `rref(B)`.

2. Bruk Python til å løse likningssystemet

$$\begin{aligned} x + 3y + 4z + 6u &= 3 \\ -x + y + 3z - 5u &= 5 \\ 2x - 3y + z + 6u &= -2 \\ 2x + 3y - 2z + u &= 0 \end{aligned}$$

3. Finn den generelle løsningen av likningssystemet

$$\begin{aligned} 2x - 3y + 4z + 5u &= 6 \\ x + \ y + \quad 3u &= 4 \\ 4x - 3y + 2z + 3u &= 5 \end{aligned}$$

ved hjelp av Python -kommandoen `rref`.

4. Skriv matrisen

$$A = \begin{pmatrix} 0 & 3 & -2 & 7 & 1 & 2 & 7 & 0 \\ 2 & 0 & 1 & -4 & 3 & 2 & 0 & 4 \\ -1 & 3 & 4 & -1 & 2 & -5 & 6 & 2 \end{pmatrix}$$

på redusert trappeform ved å la Python utføre radoperasjonene én for én (du får altså ikke lov til å bruke `rref` eller en lignende kommando). Beskriv den generelle løsningen til likningssystemet som har A som utvidet matrise.

5. a) Figuren viser spillebrettet for et enkelt terningspill. Spillerne starter på "Start" og kaster en vanlig terning for å flytte. De trenger eksakt riktig antall øyne for å gå i mål (står de på 11 og kaster en femmer, 'spretter" de altså ut til 10 ved å telle på denne måten 12-mål-12-11-10). La t_i være antall kast du må regne med å gjøre før du går i mål (dvs. det forventede antall kast) dersom du står på felt i.

Mål	12	11	10	9	8	7
Start	1	2	3	4	5	6

Forklar hvorfor

$$\begin{aligned} t_1 &= \frac{1}{6}t_2 + \frac{1}{6}t_3 + \frac{1}{6}t_4 + \frac{1}{6}t_5 + \frac{1}{6}t_6 + \frac{1}{6}t_7 + 1 \\ t_2 &= \frac{1}{6}t_3 + \frac{1}{6}t_4 + \frac{1}{6}t_5 + \frac{1}{6}t_6 + \frac{1}{6}t_7 + \frac{1}{6}t_8 + 1 \\ &\vdots \quad \vdots \quad \vdots \qquad \vdots \qquad \vdots \qquad \vdots \qquad \vdots \\ t_6 &= \frac{1}{6}t_7 + \frac{1}{6}t_8 + \frac{1}{6}t_9 + \frac{1}{6}t_{10} + \frac{1}{6}t_{11} + \frac{1}{6}t_{12} + 1 \\ t_7 &= \frac{1}{6}t_8 + \frac{1}{6}t_9 + \frac{1}{6}t_{10} + \frac{1}{6}t_{11} + \frac{1}{6}t_{12} + 1 \\ t_8 &= \frac{1}{6}t_9 + \frac{1}{6}t_{10} + \frac{1}{6}t_{11} + \frac{1}{3}t_{12} + 1 \\ t_9 &= \frac{1}{6}t_{10} + \frac{1}{3}t_{11} + \frac{1}{3}t_{12} + 1 \end{aligned}$$

$$\begin{aligned} t_{10} &= \frac{1}{6}t_{10} + \frac{1}{3}t_{11} + \frac{1}{3}t_{12} + 1 \\ t_{11} &= \frac{1}{6}t_9 + \frac{1}{6}t_{10} + \frac{1}{6}t_{11} + \frac{1}{3}t_{12} + 1 \\ t_{12} &= \frac{1}{6}t_8 + \frac{1}{6}t_5 + \frac{1}{6}t_6 + \frac{1}{6}t_{11} + \frac{1}{6}t_{12} + 1 \end{aligned}$$

Forklar videre hvorfor dette er et lineært likningssystem og bruk kommandoen `rref` til å finne løsningen. Hvor mange kast må du regne med å bruke når du står på start?

b) Løs problemet i a) når du ikke trenger eksakt antall øyne for å komme i mål.

8 Ta vare på arbeidet ditt

Vi skal nå se på noen mer datatekniske ting om hvordan du kan ta vare på det du har gjort i Python . Hvis du vil ta vare på Python -kjøringene dine med tanke på senere redigering, kan du skrive

```
%logstart -r -o filnavn
```

i et IPython Shell. Innholdet i kommandovinduet blir nå fortløpende skrevet til filen du oppga, som blir opprettet i katalogen du står i. For å avslutte dagbokføringen, skriver du

```
%logstop
```

Hvis du senere ønsker å skrive til den samme dagbokfilen, gir du kommandoen

```
%logstart -r -o filnavn append
```

Logging kan i tillegg temporært startes og stoppes (uten at du må oppgi den aktive logfila på nytt). Dette gjøres med kommandoene `%logon` og `%logoff`. Du kan senere kjøre innholdet i en lagret logfil ved å skrive

```
%runlog filnavn
```

Logfiler kan redigeres på vanlig måte, og egner seg derfor bra til obliger og lignende. Du kan gjøre de Python -kjøringene du ønsker, lagre dem i loggen, og etterpå legge til kommentarer, stryke uaktuelle utregninger osv. Vær oppmerksom på at figurer ikke lagres i loggen!

Noen ganger ønsker vi ikke å ta vare på så mye av det vi har gjort, alt vi trenger er å beholde verdien på visse variable til senere kjøringer. Skriver du

```
%store x
%store y
%store A
```

lagres verdiene til `x, y` og `A`. Når du skal lagre variable, er det ofte lurt å gi dem mer beskrivende navn enn `x, y, A`. Et variabelnavn i Python kan bestå av bokstaver og tall, men må begynne med en bokstav. Python skiller mellom store og små bokstaver. Du bør unngå å bruke innebygde funksjons- og kommandonavn på variablene dine.

Når du har arbeidet en stund, kan du fort miste oversikten over hvilke variabelnavn du har brukt. Kommandoen

```
who
```

gir deg en oversikt. Hvis du vil slette innholdet i variabelen `x`, skriver du

```
%store -d x
```

Vil du slette alle variablene, skriver du

```
%store -z
```

Oppgave til Seksjon 8

1. Lag en logfil og rediger den.

9 Programmering i Python

Skal du få virkelig nytte av Python , må du lære deg å programmere. Har du programmert i et annet språk tidligere, bør ikke dette være særlig problematisk: alt du behøver er å lære deg litt Python -syntaks. Har du aldri programmert før, står du overfor en litt større utfordring, men vi skal prøve å hjelpe deg igang.

Et program er egentlig ikke noe annet enn en sekvens av kommandoer som du vil at Python skal utføre, men det er to ting som gjør at virkelige programmer er litt mer kompliserte enn de kommandosekvensene vi hittil har sett på. Den ene er at virkelige programmer gjerne inneholder *løkker*, dvs. sekvenser av kommandoer som vi vil at maskinen skal gjennomføre mange ganger. Vi skal se på to typer løkker: *for-løkker* og *while-løkker*. Den andre tingen som skiller virkelige programmer fra enkle kommandosekvenser, er at hvilken utregning som skal utføres på et visst punkt i programmet, kan avhenge av resultatet av tidligere beregninger som programmet har gjort. Dette fanger vi opp med såkalte *if-else-setninger*.

La oss se på et typisk problem som kan løses ved hjelp av løkker. Vi skal ta for oss *Fibonacci-tallene*, dvs. den tallfølgen $\{F_n\}$ som begynner med $F_1 = 1$, $F_2 = 1$ og tilfredsstiller

$$F_n = F_{n-1} + F_{n-2} \qquad \text{for } n \geq 3 \tag{1}$$

Ethvert tall i følgen (bortsett fra de to første) er altså summen av de to foregående. Det er lett å bruke formelen til å regne ut så mange Fibonacci-tall vi vil:

$$F_3 = F_2 + F_1 = 1 + 1 = 2$$
$$F_4 = F_3 + F_2 = 2 + 1 = 3$$
$$F_5 = F_4 + F_3 = 3 + 2 = 5$$
$$F_6 = F_5 + F_4 = 5 + 3 = 8$$

og så videre. Vi ser at vi hele tiden utfører regnestykket i formel (1), men at vi for hvert nytt regnestykke oppdaterer n-verdien.

La oss nå lage et lite Python -program som regner ut de 20 første Fibonacci-tallene. Programmet gjør dette ved å lage en 20-dimensjonal vektor der F_k er den k-te komponenten.

Listing 7: Program som regner ut de 20 første Fibonacci-tallene

```
# coding=utf-8
```

```
F=[1,1] # forteller Python at F_0=1, F_1=1
for n in xrange(2,20):      # starter løkken, angir hvor langt den går
  F = F + [F[n-1]+F[n-2]]  # regner ut neste Fibonacci-tall
```

(legg merke til setningen `# coding=utf-8`, som sørger for at tegn blir tolket som UTF-8. Denne setningen er ofte droppet i kodeeksemplene i heftet). Det burde ikke være så vanskelig å skjønne hvordan programmet fungerer: det utfører de samme beregningene som vi gjorde ovenfor fra $n = 3$ til og med $n = 20$ (dvs. mellom grensene angitt i for-løkken). Vil du nå vite hva det 17. tallet i følgen er, kan du nå skrive

```
print F[16]
```

I for-løkker bestemmer vi på forhånd hvor mange ganger løkken skal gjennomløpes. Ofte ønsker vi å fortsette beregningene til et visst resultat er oppnådd uten at vi på forhånd vet hvor mange gjennomløpninger som trengs. I disse tilfellene er det lurt å bruke en while-løkke. Det neste programmet illustrerer dette ved å regne ut Fibonacci-tallene inntil de når 10 000.

Listing 8: Program som regner ut alle Fibonacci-tallene som er mindre enn 10 000.

```
from numpy import *

F=[1,1]
n=2
while F[n-1]<10000:
  F = F + [F[n-1]+F[n-2]]
  n=n+1
```

Legg merke til at i en while-løkke må vi selv oppdatere fra n til $n + 1$, mens denne oppdateringen skjer automatisk i en for-løkke. Dette skyldes at while-løkker er mer fleksible enn for-løkker, og at man også ønsker å tillate andre typer oppdatering enn at n går til $n + 1$.

La oss også se på et enkelt eksempel på bruk av en if-else-setning. Det litt tossete progammet printer ut de like Fibonacci-tallene samt indeksen til alle odde Fibonacci-tall. Legg merke til operatoren `m % k` som gir oss resten når m deles på k (dersom k er lik 2 som i programmet, blir resten 0 når m er et partall, og 1 når m er et oddetall).

```
# coding=utf-8
from numpy import *

F=[1,1]
for n in xrange(2,20):     # starter for-løkke
  F = F + [F[n-1]+F[n-2]] # regner ut neste Fibonacci-tall
  if (F[n] % 2) == 0:      # innleder if-else setningen ved å
                           # sjekke om $F(n)$ er delelig med 2
    print F[n]             # skriver ut $F(n)$ hvis det er et partall
  else:                    # innleder else-delen av setningen
    print n                # skriver ut $n$ dersom $F(n)$ er et oddetall
```

Legg merke til at dobbelte likhetstegn (==) er brukt her. I Python brukes det vanlige likhetstegnet til å tilordne verdier: skriver du

```
C=D
```

får C den (forhåpentligvis allerede definerte) verdien til D. For å sjekke om to allerede definerte verdier er like, må du derfor bruke et annet symbol, nemlig det dobbelte likhetstegnet ==. Nyttige symboler av denne typen, er

```
<       # mindre enn
<=      # mindre enn eller lik
>       # større enn
>=      # større enn eller lik
==      # lik
!=      # ikke lik
```

Når du skriver programmer, får du også bruk for logiske operatorer som *og, eller* og *ikke*. I Python skriver du dem slik:

```
and     # og (& kan også brukes)
or      # eller (| kan også brukes)
not     # ikke
```

Som et eksempel tar vi med et lite program som skriver ut Fibonacci-tall som er partall, men ikke delelig på 4:

Listing 9: Program som skriver ut Fibonacci-tallene som er partall som ikke er delelig med 4.

```
from numpy import *

F=[1,1]
for n in xrange(2,50):
  F = F + [F[n-1]+F[n-2]]
  if ((F[n] % 2)==0) and ((F[n] % 4)!=0):
    print F[n]
```

I litt større program har vi ofte behov for å avslutte en løkke og fortsette eksekveringen av den delen av programmet som kommer umiddelbart etter løkken. Til dette bruker vi kommandoen `break`. Det neste programmet illustrerer bruken. Det skriver ut alle Fibonacci-tall mindre enn 1000 (siden størrelsen på Fibonacci-tallene når 1000 før indeksen kommer til 50):

```
from numpy import *

F=[1,1]
for n in xrange(2,50):
  F = F + [F[n-1]+F[n-2]]
  if F[n]>1000:
    break
  print F[n]
```

La oss ta en nærmere titt på den generelle syntaksen for for-løkker, while-løkker, if-setninger og break-kommandoer (ikke bry deg om dette hvis du synes

det ser rart og abstrakt ut på det nåværende tidspunkt: det kan likevel hende du får nytte av det når du har fått litt mer programmeringstrening). Syntaksen til en for-løkke er:

```
for n in xrange(start,stopp,steg):
  setninger
```

Hvis steglengden er 1, trenger vi bare å oppgi start- og stoppverdiene. Syntaksen til en while-løkke er:

```
while feil < toleranse:
  setninger
```

Syntaksen for if-setninger er:

```
if n < nmaks:
  setninger
```

eller

```
if n < middel:
  setninger
elif n > middel:
  setninger
else:
  setninger
```

Syntaksen for break-kommandoer er

```
for n in xrange(antall):
  setninger
  if feil > toleranse:
    break
  setninger
```

Vi avslutter denne seksjonen med noen ord om effektivitet. I programmene ovenfor har vi begynt med å la F være 2-tuppelet `[1 1]` og så latt Python utvide lengden på tuppelet etter hvert som programmet kjører. Det viser seg at Python går fortere dersom vi gir n-tuppelet den "riktige" størrelsen fra starten av. Det første programmet vårt blir altså raskere om vi skriver det om på denne måten:

```
from numpy import *

F=zeros(20)
F[0]=1
F[1]=1
for n in xrange(2,20):
  F[n]=F[n-1]+F[n-2]
```

Inntjeningen spiller selvfølgelig ingen rolle når programmet er så kort som her, men for store utregninger kan den ha betydning.

La oss helt til slutt nevne at det egentlig er enda mer effektivt å unngå for-løkker der det er mulig; ofte kan vi erstatte dem ved å bruke vektorer og

komponentvise operasjoner isteden. Ønsker vi for eksempel å lage en vektor med de 5 første positive oddetallene som elementer, vil følgende for-løkke gjøre jobben:

```
for i in xrange(5):
  a[i]=2*i+1
```

Men det er mer effektivt å lage vektoren på følgende måte:

```
a=2*range(5)+1
```

Vi skal ikke legge vekt på effektiv programmering i dette kurset, men med tanke på senere kurs er det greit å være oppmerksom på at slik bruk av "vektor-indeksering" kan gjøre store programmer mye raskere.

I sammenligningen med andre programmeringsspråk kan vi generelt si at språk som C++ vil gjennomløpe `for`-løkker mye raskere enn Python . Man skulle kanskje tro at Python utfører operasjoner på matriser gjennom rene implementasjoner med `for`-løkker. For eksempel, matrisemultiplikasjon kan jo implementeres rett-fram med Python slik:

Listing 10: Rett-fram implementasjon av matrise-multiplikasjon

```
# coding=utf-8
from numpy import *

def mult(A,B):
  (m,n)=shape(A)
  (n1,k)=shape(B) # Må egentlig sjekke at n=n1
  C=zeros((m,k))
  for r in xrange(m):
    for s in xrange(k):
      for t in xrange(n):
        C[r,s] += A[r,t]*B[t,s]
  return C
```

Sammenligner du dette med å kjøre `A*B` i Python i stedet vil du merke stor forskjell - ihvertfall når matrisene er store. Python regner derfor ikke alltid ut multiplikasjon av store matriser helt på denne måten.

Determinanten er en annen ting som kan regnes ut på en smartere måte enn ved å følge definisjonen direkte. Et program som bruker definisjonen av determinanten direkte kan se slik ut:

Listing 11: Rett-fram implementasjon av determinanten

```
# coding=utf-8
from numpy import *

def detdef(A):
    """Funksjon som beregner determinanten til en matrise A
    rekursivt ved bruk av definisjonen av determinanten."""

    if A.shape[0] != A.shape[1]:
        print 'Feil: matrisen må være kvadratisk'
```

```
        return
    n = A.shape[0]

    # Determinanten av en 2x2-matrise kan regnes ut direkte
    if n == 2:
        return A[0,0]*A[1,1] - A[0,1]*A[1,0]
    # For større matriser bruker vi en kofaktorekspansjon
    else:
        determinant = 0.0
        for k in xrange(n):
            # Lag undermatrisen gitt ved å fjerne kolonne 0, rad k
            submatrix = vstack((A[0:k,1:n],A[k+1:n,1:n]))
            # Legg til ledd i kofaktorekspansjon
            determinant += (-1)**k * A[k,0] * detdef(submatrix)
        return determinant
```

Funksjonen `detdef` er rekursiv: den kaller seg selv helt til vi har en matrise som er så liten at vi kan regne ut determinanten direkte. Sørg her spesielt for at du forstår den delen av koden hvor $(n-1)\times(n-1)$-undermatrisen konstrueres. I koden ser du også at vi legger inn en sjekk på at matrisen er kvadratisk. Hvis den ikke er det skrives en feilmelding. Denne koden er ikke spesielt rask: Python kan regne ut determinanten mye raskere med sin innebygde metode. Determinanten til en matrise A kan du regne ut ved å skrive `linlag.det(A)` i Python . Test ut dette ved å kjøre følgende kode:

Listing 12: Sammenligning av ytelse for to funksjoner

```
from detdef import *
from numpy import *
import time

A=random.rand(9,9)
e0=time.time()
linalg.det(A)
print time.time()-e0
e0=time.time()
detdef(A)
print time.time()-e0
```

Her lages en (tilfeldig generert) 9×9-matrise, og determinanten regnes ut på to forskjellige måter. I tillegg tar Python tiden på operasjonene: `time.time()` tar tiden, og `print` skriver den ut på displayet. Kjør koden og se hvor mye raskere den innebygde determinantfunksjonen i Python er! Du kan godt prøve med en tilfeldig generert 10×10-matrise også, men da bør du være tålmodig mens du venter på at koden skal kjøre ferdig, spesielt hvis du sitter på en litt treg maskin.

Siden mye kode er tidkrevende å kjøre, kan det ofte være lurt å sette inn noen linjer med kode (for eksempel i `for`-løkker som kjøres mange ganger) som skriver ut på skjermen hvor langt programmet har kommet. Til dette kan du bruke funksjonen `print`, som over . Skriver du

```
print 'Nesten ferdig'
```

så vil programmet skrive den gitte teksten ut på skjermen . En litt mer avansert variant er

```
print 'Ferdig med', k, 'iterasjoner av løkka'
```

Hvis `k` er en løkkevariabel, så vil koden over til enhver tid holde styr på hvor langt vi er kommet i løkka.

Oppgaver til Seksjon 9

1. En følge er gitt ved $a_1 = 1$, $a_2 = 3$ og $a_{n+2} = 3a_{n+1} - 2a_n$. Skriv et program som genererer de 30 første leddene i følgen.

2. I denne oppgaven skal vi se på en modell for samspillet mellom rovdyr og byttedyr. Vi lar x_n og y_n betegne hhv. antall rovdyr og antall byttedyr etter n uker, og vi antar at

$$x_{n+1} = x_n(r + cy_n) \qquad\qquad y_{n+1} = y_n(q - dx_n)$$

der r er litt mindre enn 1, q er litt større enn 1, og c og d er to små, positive tall.

a) Forklar tankegangen bak modellen

b) Velg $r = 0.98$, $q = 1.04$, $c = 0.0002$, $d = 0.001$, $x_1 = 50$, $y_1 = 200$. Lag et Python -program som regner ut x_n og y_n for $n \leq 1000$. Plott følgene x_n og y_n i samme koordinatsystem. Hvorfor er toppene til x_n forskjøvet i forhold til toppene til y_n?

3. En dyrestamme består av tre årskull. Vi regner med at 40% av dyrene i det yngste årskullet lever videre året etter, mens 70% i det nest yngste årskullet lever videre året etter. Ingen dyr lever mer enn tre år. Et individ i det andre årskullet blir i gjennomsnitt forelder til 1.5 individer som blir født året etter. Et individ i det eldste årskullet blir i gjennomsnitt forelder til 1.4 individer som blir født året etter. La x_n, y_n, z_n være antall dyr i hvert årskull etter n år, og forklar hvorfor

$$\begin{aligned} x_{n+1} &= 1.5y_n + 1.4z_n \\ y_{n+1} &= 0.4x_n \\ z_{n+1} &= 0.7y_n \end{aligned}$$

a) Lag et Python -program som regner ut x_n, y_n og z_n for $1 \leq n \leq 100$. Plott alle tre kurvene i samme vindu. Lag et nytt vindu der du plotter alle de relative bestandene $x'_n = \frac{x_n}{x_n+y_n+z_n}$, $y'_n = \frac{y_n}{x_n+y_n+z_n}$, $z'_n = \frac{z_n}{x_n+y_n+z_n}$.

b) Gjenta punkt a), men bruk andre startverdier, f.eks. $x_1 = 300$, $y_1 = 0$, z_1=0. Sammenlign med resultatene i a). Gjør oppgavene enda en gang med et nytt sett av startverdier. Ser du et mønster?

c) La $A = \begin{pmatrix} 0 & 1.5 & 1.4 \\ 0.4 & 0 & 0 \\ 0 & 0.7 & 0 \end{pmatrix}$. Forklar at $\begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} = A^{n-1} \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix}$. Bruk dette til å regne ut x_{100}, y_{100} og z_{100} for starverdiene i a). Sammenlign med dine tidligere resultater.

10 py-filer

Hvis du skal skrive inn mange kommandoer som skal brukes flere ganger, kan det være praktisk å samle disse i en egen fil. Slike Python -filer skal ha endelsen

.py. De inneholder Python -kommandoer og kan skrives inn i en hvilken som helst editor, f.eks. emacs.

Hvis vi samler kommandoer på en fil som heter `filnavn.py`, vil Python utføre dem når vi gir kommandoen

```
run filnavn.py
```

i kommandovinduet. For at Python skal finne filen, må vi stå i samme katalog som filen er lagret i (med mindre filen ligger i søkestien til Python). Vi kan sjekke hvilken katalog vi står i med kommandoen `pwd` (present working directory) og skifte katalog med kommandoen `cd` (change directory).

Hvis vi f.eks ønsker å skrive kode som finner røttene til annengradslikningen $2x^2+3x+1=0$, kan vi legge følgende kommandoer på filen `annengradsligning.py`

Listing 13: Løsning av andregradslikningen

```
from numpy import sqrt

a=2.0
b=3.0
c=1.0
x1=(-b+sqrt(b**2-4*a*c))/(2*a)
x2=(-b-sqrt(b**2-4*a*c))/(2*a)
print 'x1', x1
print 'x2', x2
```

Når vi skriver

```
run annengradsligning.py
```

utfører Python alle kommandoene i filen og returnerer svarene

```
x1  -0.5000
x2  -1
```

En funksjon i en py-fil deklareres med nøkkelordet 'def', og har én eller flere inn- og ut-parametre. Disse parametrene er vilkårlige objekter. Variablene som brukes i funksjoner er lokale (med mindre vi definerer statiske variable).

Hvis vi ønsker å lage en funksjon som finner røttene til en vilkårlig annengradslikning $ax^2+bx+c=0$, kan vi la brukeren spesifisere likningskoeffisientene a, b og c som inn-parametre til funksjonen, og la ut-parameteren som funksjonen returnerer, være en 2-dimensjonal radvektor som inneholder røttene `r1` og `r2`. På filen `annengrad.py` legger vi da følgende funksjonskode:

Listing 14: Funksjon som løser andregradslikningen.

```
# coding=utf-8
from math import sqrt

def annengrad(a,b,c):
  if a != 0:       # sjekker at a ikke er null
    r1=(-b+sqrt(b**2-4*a*c))/(2*a)
    r2=(-b-sqrt(b**2-4*a*c))/(2*a)
  else:
```

```
    print('Koeffisienten til x**2 kan ikke være null')
  return (r1,r2)
```

Vi kan nå finne røttene til annengradslikningen $2x^2 + 3x + 1 = 0$, ved å gi kommandoen

```
(x1,x2)=annengrad(2.0,3.0,1.0)
```

Python returnerer igjen svarene $x_1 = -0.5000$ og $x_2 = -1$.

Oppgaver til Seksjon 10

1. Determinanten til en 2×2-matrise er definert ved

$$\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$$

Lag en py-fil (en funksjonsfil) som regner ut slike determinanter. Filen skal ha inn-parametre a, b, c, d og ut-parameter det $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$.

2. Skriv en py-fil (en funksjonsfil) som gir løsningen til likningssystemet

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned}$$

når det har en entydig løsning. Filen skal virke på denne måten: Inn-parametrene er koeffisientene a, b, c, d, e, f, og ut-parametrene er løsningene x, y. Dersom likningssystemet ikke har en entydig løsning, skal programmet svare: "likningssettet har ikke entydig løsning"(det behøver altså ikke å avgjøre om likningssettet er inkonsistent eller har uendelig mange løsninger).

3. Lag en funksjonsfil som regner ut leddene i følgen $\{x_n\}$ gitt ved:

$$x_{n+2} = ax_{n+1} + bx_n \qquad x_1 = c, x_2 = d$$

Filen skal ha inn-parametre a, b, c, d, m og skal returnere de m første verdiene til følgen.

4. Lag en funksjonsfil som regner ut leddene i følgen $\{x_n\}$ gitt ved:

$$x_{n+1} = ax_n(1 - x_n) \qquad x_1 = b$$

Filen skal ha inn-parametre a, b, m, og skal returnere de m første verdiene til følgen. Sett $m = 100$, se på tilfellene $b = 0.2$ og $b = 0.8$, og kjør programmet for hhv. $a = 1.5$, $a = 2.8$, $a = 3$, $a = 3.1$, $a = 3.5$, $a = 3.9$. Plott resultatene. Eksperimenter videre hvis du har lyst, men behold humøret selv om du ikke finner noe mønster; fenomenet du ser på er et av utgangspunktene for det som kalles "kaos-teori"!

11 Anonyme funksjoner og linjefunksjoner

I forrige seksjon så vi hvordan vi kunne lagre funksjoner i Python ved hjelp av funksjonsfiler. Trenger vi bare en enkel funksjon noen få ganger, er det imidlertid unødvendig komplisert å skrive og lagre en egen fil, og Python har derfor innebygd to metoder for å definere funksjoner direkte i kommanduvinduet: *anonyme funksjoner* ("anonymous functions") og *linjefunksjoner* ("one-line functions").

La oss begynne med linjefunksjoner. Hvis vi i kommandovinduet skriver

```
from math import sin
def f (x): return x*sin(1.0/x)
```

lagrer Python funksjonen $f(x) = x \sin \frac{1}{x}$. Ønsker vi å vite hva $f(5)$ er, kan vi derfor skrive

```
>>> f(5)
0.99334665397530608
```

Ønsker vi å plotte grafen til f, går vi frem på vanlig måte:

```
from numpy import *
from scitools.easyviz import *

x=arange(0.01,1,0.01,float)
y=f(x)
plot(x,y)
```

Metoden fungerer også på funksjoner av flere variable:

```
>>> def f (x,y): return y*sin(1.0/x)
>>> f(2,3)
1.438276615812609
```

forteller oss at funksjonsverdien til funksjonen $f(x, y) = y \sin \frac{1}{x}$ i punktet $(2, 3)$ er 1.4383. Legge merke til at i sitt svar på kommandoen

```
def f (x,y): return y*sin(1.0/x)
```

har Python ordnet variablene slik at vi vet at x er første og y er annen variabel. Vil vi tegne grafen til f, skriver vi

```
r=arange(0.01,1,0.01,float)
s=arange(0.01,1,0.01,float)
x,y=meshgrid(x,y,sparse=False,indexing='ij')
z=f(x,y)
mesh(x,y,z)
```

La oss nå gå over til anonyme funskjoner. Disse definerer du på denne måten:

```
g= lambda x,y : x**2*y+y*3
```

Funksjonen $g(x, y) = x^2y + y^3$ er nå lastet inn. Du kan regne ut verdier og tegne grafer på samme måte som ovenfor:

```
>>> g(1,-2)
-8
>>> r=arange(0.01,1,0.01,float)
>>> s=arange(0.01,1,0.01,float)
>>> x,y=meshgrid(r,s,sparse=False,indexing='ij')
>>> z=g(x,y)
>>> mesh(x,y,z)
```

Anonyme funksjoner er nyttige når du skal integrere funksjoner. Vil du f.eks. regne ut integralet $\int_0^2 e^{-\frac{x^2}{2}}\,dx$, så kan du bruke metoden `trapezoidal` som du programmerte i INF1100 (numerisk integrasjon ved hjelp av trapesmetoden), og skrive

```
>>> trapezoidal( lambda (x) : exp(-x**2),0,2,100)
0.88207894884004279
```

Her har vi brukt trapesmetoden med 100 punkter. Utelater du `lambda (x)`, får du en feilmelding. Det går imidlertid greit å bruke denne syntaksen til å integrere en linjefunksjon du allerede har definert:

```
>>> def h (x): return exp(-x**2)
>>> trapezoidal( h,0,2,100)
0.88207894884004279
```

men kommandoen fungerer fortsatt ikke dersom h er definert som en anonym funksjon! Vi kan bruke anonyme funksjoner i to variable sammen med funksjonen `integrate2D` som vi programmerte i INF1100 til å beregne dobbeltintegraler.

```
integrate2D(lambda x,y: x**2*y,0,1,-1,2,100,100)
```

beregner dobbeltintegralet av funksjonen x^2y over rektangelet $[0,1]\times[-1,2]$, med 100 delepunkter langs x- og y-aksen. Svaret blir 0.5, med de fire første desimalene.

Oppgaver til Seksjon 11

1. Definer $f(x)=\frac{\sin x}{x}$ som en linjefunksjon. Tegn grafen til f og regn ut integralet f fra $\frac{1}{2}$ til 2.

2. Gjenta oppgave 1, med definer nå f som en anonym funksjon.

3. Definer $f(x,y)=x^2y-y^2$ som en linjefunksjon og tegn grafen.

4. Gjenta oppgave 3, med definer nå f som en anonym funksjon.

12 Python og bilder*

Python er et utmerket verktøy til å analysere og gjøre operasjoner på bilder. De tre viktigste funksjonene er

imread `A = imread('filnavn.fmt')` leser inn et bilde med angitt filnavn og format, og lagrer det som en matrise A. ('fmt') kan være et hvilket som helst format som din versjon av Python støtter.

imshow Kommandoen `imageview(A)` lager et objekt fra matrisen A som kan vises frem som et bilde.

show viser objektet (bildet) du laget ved hjelp av kommandoen over. Fra menyene i denne dialogen kan du blant annet lagre bildet til fil.

Disse kommandoene blir tilgjengelige i Python hvis du skriver

```
from pylab import imread, imshow, show
```

For å gjøre operasjoner på bilder trenger vi også konvertere verdiene til `double`. Vi må altså skrive

```
from pylab import imread
from numpy import *

img = imread('mm.gif')
img = double(img)
```

Verdiene i matrisen `img` ligger mellom 0 og 255. Størrelsen på bildet finner du ved å skrive

```
(m,n)=shape(img)
```

12.1 Kontrastjustering

Som et eksempel på en operasjon vi kan gjøre på et bilde, la oss se på kontrastjustering. Pikselverdiene justeres da ved hjelp av en funksjon som avbilder $[0,1]$ på $[0,1]$, en såkalt kontrastjusterende funksjon. Eksempler på kontrastjusterende funksjoner er

$$\begin{aligned} f(x) &= \frac{\log(x+\varepsilon)-\log(\varepsilon)}{\log(1+\varepsilon)-\log(\varepsilon)}, \\ g(x) &= \frac{\arctan(n(x-1/2))}{2\arctan(n/2)}+1/2. \end{aligned}$$

For å bruke en kontrastjusterende funksjon må vi i tillegg avbilde pikselverdiene fra [0,255] til [0,1], og til slutt avbilde de fra [0,1] til [0,255]. Kode for kontrastjustering med $f(x)$ blir da seende slik ut:

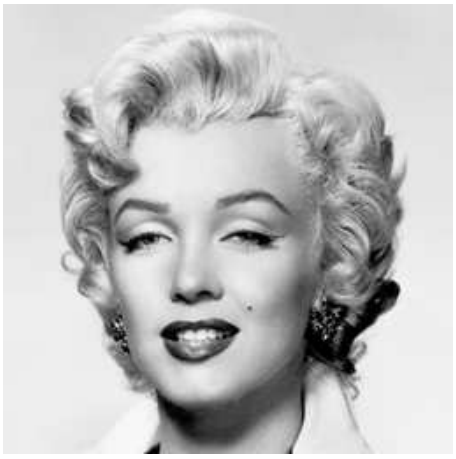
Listing 15: Kontrastjustering av et bilde

```
# coding=utf-8
from numpy import *

epsilon = 0.001             # Prov ogsaa 0.1, 0.01, 0.001
newimg = double(img)/255    # Avbilder pikselverdiene paa [0,1]
newimg = (log(newimg+epsilon) - log(epsilon))/\
         (log(1+epsilon)-log(epsilon));
newimg = newimg*255 # Avbilder verdiene tilbake til [0,255]
```

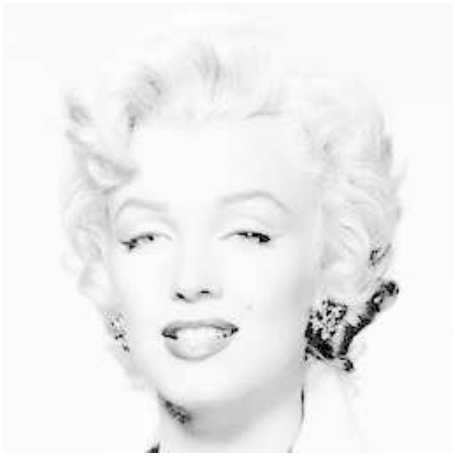
Et bilde, sammen med de kontrastjusterte versjoner med $f(x)$ og $g(x)$ ser du i Figur 1.

12.2 Utglatting

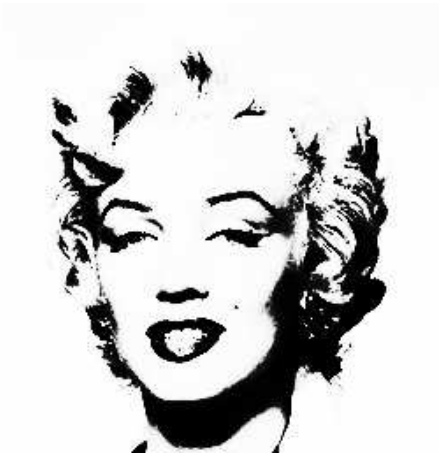
En annen vanlig operasjon på bilder er utglatting. Ved utglatting erstattes verdiene i bildet med vektede middelverdier av de nærliggende pikslene i bildet. Vektingen er definert ved en matrise, kalt en vektmatrise (eller konvolusjonskjerne), der "midterste"verdi i matrisen bestemmer vekten for det aktuelle pikselet, verdien rett over denne bestemmer vekten for pikselet rett over det aktuelle pikselet, o.s.v. Summen av alle elementene i vektmatrisen er vanligvis 1. Utglatting med en vektmatrise kan programmeres slik:



(a) Originalt



(b) Kontrastjustert med $f(x)$



(c) Kontrastjustert med $g(x)$

Figur 1: Et bilde med og uten kontrastjustering.

Listing 16: Utglatting av et bilde med en vektmatrise

```
# coding=utf-8
from numpy import *

def smooth(img,vektmatrise):
  (m,n)=shape(img)
  newimg = zeros((m,n))
  (k,k1) = shape(vektmatrise) # Vi skal ha k==k1, odde
  sc = (k+1)/2
  for m1 in xrange(m):
    for n1 in xrange(n):
      slidingwdw = zeros((k,k))
        # slidingwdw er den delen av bildet som
        # vektmatrisen blir anvendt pïf¡ for piksel (m1,n1)
      slidingwdw[ max(sc-1-m1,0):(min(sc+m-2-m1,k-1)+1) ,
                  max(sc-1-n1,0):(min(sc+n-2-n1,k-1)+1)] = \
      img[max(0,m1-(sc-1)):(min(m-1,m1+(sc-1))+1) ,
          max(0,n1-(sc-1)):(min(n-1,n1+(sc-1))+1)]
      newimg[m1,n1] = sum(vektmatrise * slidingwdw)

  return newimg
```

Inne i `for`-løkken ser du kode som kan virke litt kryptisk. Koden skyldes at i ytterkantene av bildet er det ikke nok piksler til å matche størrelsen på vektmatrisen, slik at litt ekstra kode må til for å håndtere dette. Utglatting med matrisen

$$A_1 = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

kan vi nå gjøre slik

```
from numpy import *
from smooth import *

vektmatrise=(1.0/16)*array([[1.0,2.0,1.0],[2.0,4.0,2.0],[1.0,2.0,1.0]])
newimg = smooth(img,vektmatrise)
```

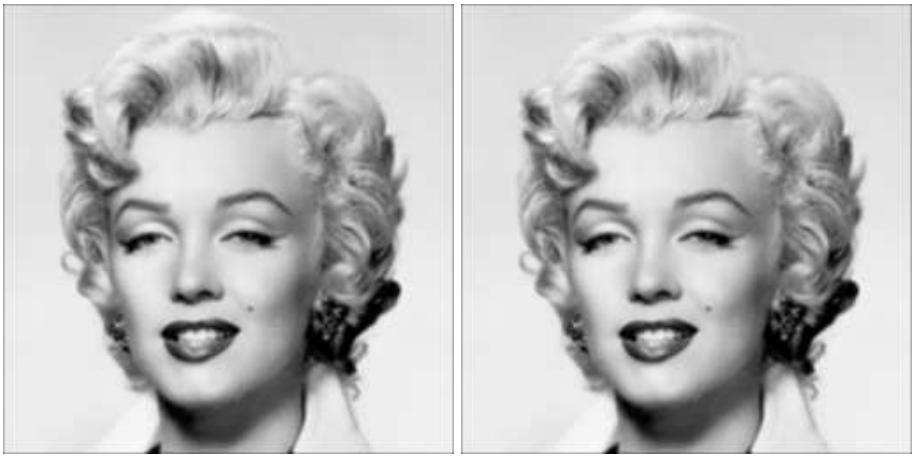
I Figur 2 viser vi bildet utglattet med vektmatrisen A_1, samt vektmatrisen

$$A_2 = \frac{1}{1024}\begin{bmatrix} 1 & 6 & 15 & 20 & 15 & 6 & 1 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 20 & 120 & 300 & 400 & 300 & 120 & 20 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{bmatrix}$$

Som du ser gir den større vektmatrisen A_2 en større utglattingseffekt.

12.3 Gradienten til et bilde

Ved å erstatte vektmatrisen med andre matriser kan vi regne ut andre interessante bilder, for eksempel som viser hvordan bildet forandrer seg i x-retningen, eller i y-



(a) Utglatting med A_1 (b) Utglatting med A_2

Figur 2: Utglatting av bildet vårt med to forskjellige vektmatriser.

retningen. Mer presist ville vi bruke

$$\texttt{vektmatrisex} = \begin{bmatrix} 0 & 0 & 0 \\ -1/2 & 0 & 1/2 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\texttt{vektmatrisey} = \begin{bmatrix} 0 & 1/2 & 0 \\ 0 & 0 & 0 \\ 0 & -1/2 & 0 \end{bmatrix}.$$

Disse svarer til approksimasjoner av $\frac{\partial}{\partial x}$ og $\frac{\partial}{\partial y}$, respektive, anvendt på bildet. Gradienten til et bilde kan vi regne ut slik:

Listing 17: Utregning av gradienten til et bilde

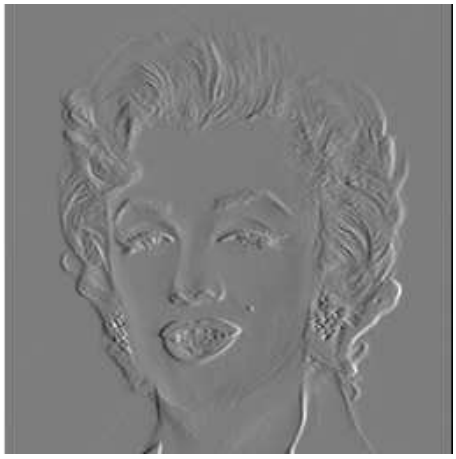
```
from numpy import *
from smooth import *

newimg = sqrt( smooth(img,vektmatrisex)**2 + \
               smooth(img,vektmatrisey)**2 )
```

Med denne koden kan det hende at de nye pikselverdiene faktisk ligger utenfor $[0, 255]$. Dette kan vi løse ved å avbilde verdiene til $[0, 1]$ ved hjelp av følgende kode:

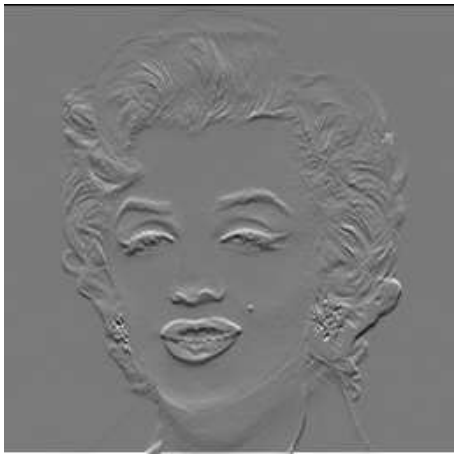
Listing 18: .]Avbildning av verdiene i et bilde på [0,1].

```
def mapto01(img):
  minval = img.min()
  maxval = img.max()
  newimg = (img - minval)/(maxval-minval)
  return newimg
```

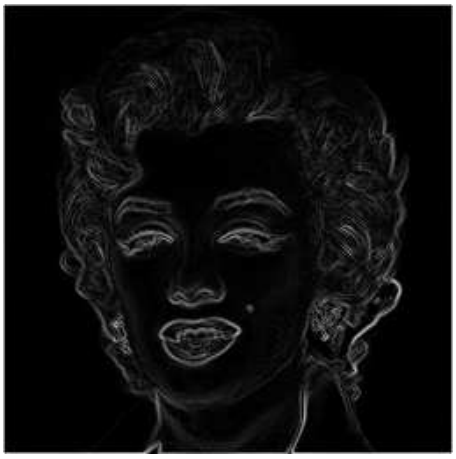
Deretter kan vi avbilde disse verdiene tilbake på $[0, 1]$. I Figur 3 har vi vist hvordan de partielt deriverte bildene og gradientbildet da blir seende ut.



(a) x-partielt deriverte av bildet



(b) y-partielt deriverte av bildet



(c) Gradienten til bildet

Figur 3: De partielt deriverte og gradienten av bildet

12.4 Eksempler på andre operasjoner på bilder

Det er også mange andre operasjoner enn kontrastjustering og utglatting som kan gjøres enkelt i Python . Her er noen flere eksempler:

Listing 19: Speiling av et bilde opp ned.

```
(m,n)=img.shape
imgupdown = zeros((m,n))
for k in xrange(m):
  imgupdown[k,:] = img[m-1-k,:]
```

Listing 20: Speiling av et bilde fra venstre til høyre.

```
(m,n)=shape(img)
imgleftright = zeros((m,n))
for k in xrange(n):
  imgleftright[:,k] = img[:,n-1-k]
```

Listing 21: Nedskalering av et bilde med faktor 2^r.

```
from numpy import *

(m,n)=shape(img)
scimg=zeros((m/2**r,n/2**r))
for m1 in xrange(m/2**r):
  for n1 in xrange(n/2**r):
    xstart = (m1-1)*(2**r)+1
    ystart = (n1-1)*(2**r)+1
    # Nedskalering skjer ved at blokker i bildet blir
    # erstattet med et piksel med verdi lik middelverdien
    # i blokken:
    scimg[m1,n1] = sum(img[xstart:(xstart+2**r),
                           ystart:(ystart+2**r)])/(2**(2*r))
```