Kapittel 1

Innføring i Python (i samarbeid med Øyvind Ryan)

Dette appendikset gir en kort innføring i Python med tanke på behovene i denne boken. Hensikten er å gi deg litt starthjelp slik at du kommer i gang med enkel programmering Avansert bruk må du lære deg andre steder (se lenkene på kursets hjemmeside). Du kan også få hjelp fra 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.

Navnet Python stammer fra TV-serien Monty Python's Flying Circus, ikke fra pythonslangen, som mange tror. Et viktig mål hos de som har utviklet Python er at det skal være morsomt å bruke, på samme måte som TV-serien er morsom. Dette ser man blant annet ved at mange tutorials og kodeeksempler i Python har referanser til Monty Python serien. For eksempel, "spam" og "eggs" er ofte brukte variabelnavn i Python, og disse er inspirert fra TV-serien.

I de første seksjonene i notatet (1-7) konsentrerer vi oss hovedsakelig om hvordan du kan 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, og hvordan du lager dine egne 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.

I dette appendikset se mange kodeeksempler. Vi har prøvd å gi mange kommentarer i koden, slik at det skal være mulig å lese seg til hva som skjer. I koden vil du se at en kommentar begynner med `#`. Det tolkes 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. Det er en god vane å legge inn forklarende kommentarer i koden. Du vil se at funksjoner i dette heftet inneholder noen kommentarer helt i starten på hvordan input og output for funksjonen skal tolkes. Dette er helt vanlig programmeringsskikk i mange språk. Kode for alle funksjonene i appendikset kan også finnes på kurssidene i filer med samme navn som funksjonen.

1.1 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 returverdiene fra dem. Etter promptet kan du skrive en kommando du vil ha utført. Skriver du f.eks.

```
>>> 3+4
```

og skifter linje, blir svaret

```
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
3/4
3**4
```

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

```
8**2/3
```

noe annet enn

```
8**(2/3)
```

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)
```

blir svaret sinus til 0.94 radianer. Vær nøye med multiplikasjonstegn: Skriver du `15sin(0.94)`, blir svaret 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 kan det være greit å vite at kvadratrotfunksjonen heter `sqrt` og at absoluttverdifunksjonen heter `abs`. Skriver du

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

regnes ut $e^{\sqrt{|-0.7|}}$. De andre standardfunksjonene heter det du er vant til: `sin, cos, tan, log`. I tillegg heter arcusfunksjonene `arcsin`, `arccos`, `arctan`.

Vår gamle venn π kalles for `pi`. Kommandoen

```
pi
```

gir den numeriske tilnærmingen 3.14159265... til π.

Regner du ut `sin(pi)` før du deg kanskje en overraskelse. Istedenfor 0 blir svaret `1.2246e-16`. Dette skyldes at beregninger primært skjer med avrundede tall.

Det finnes også funksjoner som runder av tall på flere måter:

```
floor(a)  # Runder ned til naermeste hele tall,
ceil(a)   # Runder opp til naermeste hele tall,
round(a)  # Runder av til naermeste hele tall.
```

Skal du bruke et tall flere ganger i en beregning, kan det være greit å gi det et navn. Tilordner du først verdier til variabelene `a` og `b`, kan du gange dem sammen slik:

```
a=0.3124
b=2.41
a*b
```

La oss avslutte denne innledende seksjonen med noen knep det kan være greit å vite om (blir det for mye på en gang, får du heller komme tilbake til denne delen senere). Du kan ikke endre på en kommando som allerede 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).

Oppgaver til seksjon 1.1

1 *Regn ut:* $2.2+4.7$, $5/7$, 3^2, $\frac{2\cdot 3-4^2}{13-2.2^2}$

2 *Regn 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* $\sin x^2 y$.

1.2 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. Husk at forskjellige moduler i Python kan implementere funksjoner med like navn. Hvis du skal bruke en funksjon i modulen `numpy`, og denne funksjonen også finnes i `math`, så er det viktig at du importerer `numpy` etter at du importerer `math`: Python vil velge funksjonen fra den siste modulen som du importerte. 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

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

skriver vi

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

(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]])
```

Regneoperasjoner for matriser utføres med enkle kommandoer. Har du lagt inn matrisene A og B, kan du skrive vil kommandoene

```
A+B   # regner ut summen A+B
A-B   # regner ut differensen A-B
A*B   # regner ut produktet AB
```

Dette forutsetter at matrisene har riktige dimensjoner slik at regneoperasjonene er definert (hvis ikke blir svaret en feilmelding av typen: `Objects are not aligned`). Eksempler på andre viktige kommandoer er (noen av disse befinner seg i pakken `numpy.linalg`)

```
3*A  # Ganger matrisen A med tallet 3
A**7 # Regner ut A i sjuende potens (A ganget med seg selv 7 ganger)
A.T  # Finner den transponerte av A
linalg.inv(A)      # Finner den inverse matrisen
linalg.det(A)      # Regner ut determinanten til A
D,V = linalg.eig(A) # Finner egenverdier og egenvektorer til A
                    # Søylene i matrisen V er A's egenvektorer
                    # D er et array med egenverdiene til A
                    # Egenvektorer/egenverdier kommer i samme
                    # rekkefølge:
                    # egenverdi 1 tilhører egenvektor 1 osv.
```

Her er et eksempel på en kjøring:

```
>>> from numpy import *
>>> A = matrix([[3,1,2],     # taster inn matrisen A
                [3,0,4],
```

```
                [2,1,4]])
>>> u,v = linalg.eig(A)      # finn egenvektorer og -verdier
>>> print(u)                 # skriv ut u
[ 6.6208359   1.43309508 -1.05393098]
>>> print(v)                 # skriv 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

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

med egenverdier henholdsvis 6.6208, 1.4331 og -1.0539. Vær oppmerksom på at egenvektorene alltid normaliseres 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`, og `eig`) forutsetter at A er en kvadratisk matrise.

Rangen til en matrise får du ved hjelp av funksjonen `linalg.matrix_rank`, som du kan finne i `numpy`. Vi er vant til å radredusere matrisen A og telle antall pivotsøyler for å finne rangen. Når vi setter en datamaskin til å gjøre utregningene av ramgen ved radreduksjon, vil ofte avrundingsfeil kunne føre til at det ser ut som om matrisen har full rang, selv om det egentlig ikke er tilfelle. I stedet kan datamaskinen beregne rangen ved å telle antall *singulærverdier* som er ekte større enn null! (I MAT1120 lærer vi ranger er lik antall positive singulærverdier σ_j til A). Python bruker denne fremgangsmåten, og antar at $\sigma_j > 0$ dersom $\sigma_j > \epsilon$ for en angitt toleranse $\epsilon > 0$. Dersom man ønsker å forandre den forhåndsdefinerte toleransen ϵ kan man angi dette ved å sende med en ekstra parameter til kommandoen ved å skrive

```
r = linalg.matrix_rank(A,tol=epsilon)
```

hvor ϵ er den ønskede toleransen.

Vektorer blir oppfattet 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 er det viktig å ha tenkt igjennom om du ønsker radvektorer eller søylevektorer; du kan ikke regne med at det blir forstått at du mener en søylevektor når du skriver en radvektor! Legg merke til at hvis

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

er en radvektor, vil den transponerte være søylevektoren

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

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

```
A*c
```

Vi 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 tre-dimensjonale.

Oppgaver til seksjon 1.2

1 *La*

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

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*

```
arccos(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.

1.3 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

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

er

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

Selv om slike komponentvise operasjoner ikke brukes mye i lineær algebra, er de viktige i Python siden Python bruker matriser til mer enn tradisjonell matriseregning. 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 vi få regnet 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:

```
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. Det finnes 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. Koden nedenfor forenkler dette:

```
a=range(1,101)   # Definerer a til vektoren (1,2,3,..., 100)
a=range(1,101,2) # Definerer a til vektoren med alle oddetall < 100
a=range(b,c,h)   # Definerer a til vektoren med b forst,
                 # deretter b+h, b+2h osv. inntil vi kommer til c.
                 # Siste komponent er storste tallet paa formen
                 # a+nh smindre enn c.
```

Skal vi regne ut summen av kvadratene av alle partall opp til 100, kan vi altså skrive

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

Kommandoen `linspace` 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.

Komponentvise operasjoner blir også brukt når vi regner ut integraler. $\int_0^2 e^{-\frac{x^2}{2}}dx$ kan regnes ut ved å skrive

```
from scipy.integrate import quad
from math import *

prin(quad(lambda x : exp(-x**2),0,2))
```

`quad` står for *quadrature*, et gammelt navn for integrasjon. Denne ble importert fra pakken `scipy.integrate`.

Oppgaver til seksjon 1.3

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

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

3 *Legg inn* $(0, \frac{1}{100}, \frac{2}{100}, \ldots, 1)$*. 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 *Regn ut produktet* $\frac{1}{2} \cdot \frac{3}{4} \cdots \frac{99}{100}$*.*

1.4 Grafer

For å plotte trenger vi å importene pakken `matplotlib.pyplot`:

```
import matplotlib.pyplot as plt
```

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 kan brukes på flere måter til å tegne kurver:

```
plt.plot(x,y)  # lager plott der (x_1,y_1) forbindes med (x_2,y_2),
               # (x_2,y_2) forbindes med (x_3,y_3) osv.
               # Forbindelselinjene er heltrukne linjer.
plt.figure()
plt.plot(x)    # som plot(x,y), men i stedet med punktene
               # (1,x_1),(2,x_2), (3,x_3) osv.
```

`plt.figure()` aktiverer et nytt vindu slik at de to plottene kommer i forskjellige vinduer: Vi må også skrive

```
plt.show()
```

for at plottene skal vises i et vindu. Vær oppmerksom på at dette vinduet kan gjemme seg bak andre vinduer! Når du er ferdig med et plott bør du også skrive

```
plt.close()
```

for å frigjøre ressurser knyttet til plottet.

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. Hvis vi skal tegne grafen til $\sin \frac{1}{x}$ over intervallet $[-\pi, \pi]$ velger vi først et tuppel som gir oppdeling av intervallet i mange små biter, deretter velger vi tuppelet av tilhørende funksjonsverdier, før vi til slutt plotter punktene

```
x=linspace(-pi,pi,100)
y=sin(1/x)
plt.plot(x,y)                # plotter punktene
```

Dropper du å ta med `plt.figure()` vil plottene som kommer vises i samme vindu. Plott av $z = x^2 \sin \frac{1}{x}$ i samme vfindu får vi derfor ved å legge til

```
z=x**2*sin(1.0/x)
plt.plot(x,z)
```

Nå som vi har flere grafer i samme vindu kan det være lett å gå i surr på hva som er hva. Skriver du

```
plt.legend(['Dette er den første grafen','Dette er den andre grafen'])
```

vil hver graf bli angitt med den assosierte teksten, slik at de kan skilles fra hverandre. Grafene du lager vil fortsette å komme i samme vindu inntil du gir kommandoen

```
plt.figure()
```

Du kan navngi forskjellige vinduer slik at du kan hoppe mellom dem ved å angi en identifikator for vinduene:

```
plt.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

```
plt.axis([-pi, pi, -2.5, 2.5]) # Setter grensene paa aksene
plt.axis('square')             # Gir et kvadratisk plottevindu
plt.axis('equal')              # Gir deg samme maalestokk på begge aksene.
```

Du kan gi figuren en tittel, og sette navn på aksene ved å skrive

```
plt.title('Grafen til en vakker funksjon')
plt.xlabel('x-akse')
plt.ylabel('y-akse')
```

Ønsker du tekst på figuren, kan du skrive

```
plt.text(a, b, 'tekst')
```

der (a, b) er koordinatene til det punktet på figuren der teksten skal begynne. Det er lett å angi farge på grafene, samt måten grafen skal tegnes:

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

```
plt.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 i pdf-format. Dette kan du gjøre slik:

```
plt.savefig('figur.pdf', format='pdf')
```

Her kan pdf byttes ut med andre støttede formater.

Oppgaver til seksjon 1.4

1 *Legg inn 6-tuplene $a = (3, 1, -2, 5, 4, 3)$ og $b = (4, 1, -1, 5, 3, 1)$ 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 *Tegn 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 *Tegn 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 $\frac{1}{10000}$.*

1.5 Tre-dimensjonale grafer

Det finnes flere kommandoer som kan brukes til å lage tre-dimensjonale 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:

```
import matplotlib.pyplot as plt
from math import *
from numpy import *
from mpl_toolkits.mplot3d import Axes3D

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 vaare
z=x*y*sin(x*y)                  # Regn ut funksjonsverdiene
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(x, y, z)                    # selve plottingen
```

For tredimensjonal plotting fikk vi bruk for pakken `mpl_toolkits.mplot3d`. Grafen kommer opp i et eget figurvindu akkurat som for to-dimensjonale figurer. Ønsker du bare å få ut nivåkurvene til en flate, kan du bruke kommandoen `contour`. Denne kommer i flere varianter:

```
ax.contour(x,y,z)          # Programmet vil her selv velge hvor
                           # mange nivåkurver som blir tegnet
ax.contour(x,y,z,levels=v) # v er en vektor som inneholder
                           # verdiene til nivåkurvene vi skal tegne
ax.contour(x,y,z,n)        # n er antall nivåkurver du onsker
```

Parameteren som bestemmer antall nivåkurver, eller nivåene til disse, er ofte nødvendig, siden programmet ikke alltid klarer å finne de mest interessant nivåkurvene. Det kan også tenkes at vi er interessert i helt konkrete nivåer, som programmet umulig kan gjette på. Med kommandoen `clabel` får du også skrevet nivået til nivåkurvene på grafen:

```
cs = ax.contour(x,y,z)
plt.clabel(cs)
```

Normalt vil nivåkurvene bli tegnet i forskjellige farger. Dette er nyttig på skjermen, men kan være mindre praktisk dersom du ønsker å lime figuren inn i et svart-hvitt dokument. Skriver du

```
ax.contour(x,y,z,8,colors='k')
```

får du 8 nivåkurver tegnet i svart ('k' er symbolet for svart farge).

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
ax.plot(x,y,z)
```

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

Det finnes også støtte for å tegne vektorfelt. Skriver du

```
ax.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). Hvis du dropper x og y parametrene, så vil vektorene tegnes opp i et standard rutenett der alle punkter har heltallige koordinater større enn 0.

Når vi tegner vektorfelt kan det være lurt å ikke bruke for mange punkter (x, y), siden det fort tegnes så mange vektorer at disse kolliderer med hverandre.

Oppgaver til seksjon 1.5

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

2 *Tegn 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* `plot` *til å lage en tre-dimensjonal strektegning av en terning.*

1.6 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):

```
A.sum(0)      # Returnerer en radvektor der hvert element er
              # summen av tilsvarende soyle
A.sum(1)      # Returnerer en soylevektor 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):

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

På samme måte kan vi regne ut maksimum av elementene i en matrise på forskjellige måter:

```
A.amax(0)     # Returnerer en radvektor der hvert element er
              # maksimum av tilsvarende soyle
A.amax(1)     # Returnerer en soylevektor der hvert element er
              # maksimum av tilsvarende rad
A.amax()      # Returnerer maksimum av alle elementene i matrisen
```

Minimum går på samme måte. Vi har også funksjonene `max` og `min`, som virker litt annerledes.

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

Du har 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=concatenate((A,B),axis=1)
```

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

```
A[0,1]    # skriver ut elementet i forste rad, andre soyle i A
A[0,:]    # skriver ut hele den forste raden i A
A[:,1]    # skriver ut hele den andre soylen i A
```

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

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

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. Legg merke til den noe spesielle formen her, der kommandoen `ix_` blir kalt. 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.

Det finnes mange andre spesialkommandoer for å manipulere matriser. Her følger noen kommandoer for å generere matriser 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
triu(A)            # Lager en ovre triangulaer matrise fra A, d.v.s. alle
                   # elementene under diagonalen er blir satt til 0
tril(A)            # Lager en nedre triangulaer matrise fra A, d.v.s. alle
                   # elementene over diagonalen er blir satt til 0
random.permutation(5) # Lager tilfeldig permutasjon av tall fra 1 til 5
```

Som et enkelt eksempel tar vi med at

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

gir matrisen

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

Oppgaver til seksjon 1.6

1 *Skriv inn matrisene*

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

og

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

og gjennomfør operasjonene

```
C=concatenate((A,B),axis=1)
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[[0,2],:]=4*C[[0,2],:]
```

3 *Undersøk hva kommandoen*

```
concatenate((A,B),axis=0)
```

gjør når A og B er to matriser.

4 *Bruk kommandoene*

```
random.rand(2,2)
random.rand(3,3)
random.rand(4,4)
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?

1.7 Radoperasjoner

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.

Det finnes en egen kommando som bringer en symbolsk matriser på redusert trappeform. Den heter `rref` (for *reduced row echelon form*), og finnes i modulen `sympy`.

```
from sympy import *
```

Starter vi med matrisen

```
A = Matrix([[1,4,-5,7,3,2,8,-1],
            [2,3,-2,5,3,3,7,8],
            [-1,1,2,3,-1,4,4,4]])
```

leder kommandoen

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

til den reduserte trappeformen

```
Matrix([
[1, 0, 0, -12/25, 22/25, -1/5, -1/25, 84/25],
[0, 1, 0,  53/25,  7/25,  9/5, 69/25, 54/25],
[0, 0, 1,    1/5,  -1/5,    1,   3/5,  13/5]])
```

Du kan også 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,:]
```

blir svaret

```
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[2,:]=C[2,:]+C[0,;]
```

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

(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 kommandoen `rref` er innebygd. Man støter imidlertid ofte på matriser med spesiell struktur, f.eks. svært mange nuller. Da er det ofte mer effektivt selv å programmere hvilke radoperasjoner som skal gjøres, enn å kjøre en standardkommando som `rref` som ikke tar hensyn til strukturen til matrisene.

Radoperasjoner kan brukes til å løse lineære likningssystemer, men du har også muligheten til å finne løsningen med én kommando. Dersom A er en ikkesingulær, 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 over svaret

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

Oppgaver til seksjon 1.7

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

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

og gjennomfør operasjonene `A.rref()` *og* `B.rref()`.

2 *Løs 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*

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

ved hjelp av 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 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

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

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

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.*

1.8 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. Hvis du vil ta vare på 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 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 kan bestå av bokstaver og tall, men må begynne med en bokstav. Det skilles mellom store og små bokstaver. Du bør unngå å bruke innebygde funksjons- og kommandonavn på variabelene 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 variabelene, skriver du

```
%store -z
```

Oppgaver til seksjon 1.8

1 *Lag en logfil og rediger den.*

1.9 Programmering

Skal du gjøre mer avanserte ting må du lære deg å programmere. Har du programmert i et annet språk tidligere bør ikke dette være særlig problematisk, siden syntaksen sannsynligvis ligner mye på det du kan fra før. Har du aldri programmert før, står du overfor en litt større utfordring. Det finnes flere innebygde hjelpefunksjoner som kan hjelpe deg. Blant annet har de fleste Shell en autokompletteringsfunksjon for kode, slik at forslag til navn på funksjoner blir listet opp for deg hvis du ikke helt husker dem selv. Mange Shell hjelper deg også med å sørge for at du setter opp nestede parenteser riktig ved kall på funksjoner.

Et program er egentlig ikke noe annet enn en sekvens av kommandoer som du vil ha utført, 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

$$F_n = F_{n-1} + F_{n-2} \qquad \text{for } n \geq 3 \tag{1.1}$$

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:

$$\begin{aligned}
F_3 &= F_2 + F_1 = 1 + 1 = 2 \\
F_4 &= F_3 + F_2 = 2 + 1 = 3 \\
F_5 &= F_4 + F_3 = 3 + 2 = 5 \\
F_6 &= F_5 + F_4 = 5 + 3 = 8
\end{aligned}$$

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 program som regner ut de 20 første Fibonacci-tallene. Programmet gjør dette ved å lage en 20-dimensjonal vektor der F_k er komponent k:

```
# coding=utf-8

F=[1,1] # sier til Python at F_0=1, F_1=1
for n in range(2,20):     # starter lokka, angir hvor langt den gaar
  F = F + [F[n-1]+F[n-2]]  # regner ut neste Fibonaccitall
```

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. Legg også merke til indenteringen i innmaten av `for`-løkka, som er helt sentral i Python. 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:

```
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 range(2,20):    # starter for-lokke
  F = F + [F[n-1]+F[n-2]] # regner ut neste Fibonacci-tall
  if (F[n] % 2) == 0:     # innleder if-else setningen ved aa
                          # 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. Det vanlige likhetstegnet brukes 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
>       # storre enn
>=      # storre enn eller lik
==      # lik
!=      # ikke lik
```

Når du skriver programmer, får du også bruk for logiske operatorer som `og`, `eller` og `ikke`. Du skriver dem slik:

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

Disse operatorene returnerer de logiske verdien `true` og `false`. Som et eksempel tar vi med et lite program som skriver ut Fibonacci-tall som er partall, men ikke delelig på 4:

```
from numpy import *

F=[1,1]
for n in range(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 range(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 range(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 range(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 utvidet lengden på tuppelet etter hvert som programmet kjører. Det viser seg at programmet 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 range(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 range(5):
  a[i]=2*i+1
```

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

```
a=2*arange(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-indekseringkan 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 operasjoner på matriser utføres gjennom rene implementasjoner med `for`-løkker. For eksempel, matrisemultiplikasjon kan jo implementeres rett-fram slik:

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

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

Sammenligner du dette med å kjøre `A*B` 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:

```
# 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 range(n):
            # Lag undermatrisen gitt ved å fjerne kolonne 0, rad k
            submatrix = concatenate((A[0:k,1:n],A[k+1:n,1:n]),axis=0)
            # 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: determinanten kan regnes ut mye raskere med en innebygd metode. Determinanten til en matrise A kan du regne ut ved å skrive `linlag.det(A)` Test ut dette ved å kjøre følgende kode:

```
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 tas tiden på operasjonene: `time.time()` tar tiden, og `print` skriver den ut på displayet. Kjør koden og se hvor mye raskere den innebygde determinantfunksjonen 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`. 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.

1.10 Input og output

Noen ganger trenger vi å skrive et program som er avhengig av input fra brukeren. Et eksempel er et program som kjører mange ting, og man vil av og til at brukeren skal få anledning til å avbryte programmet. Dette kan gjøres ved hjelp av kommandoen `input`:

```
svar=input('Vi du at programmet skal fortsette (J/N)? ')
if svar=='J': # Sjekker om svaret fra brukeren er 'J'
    # Fortsett aa kjore programmet, brukeren ville fortsette
elif svar == 'N':
    # Avbryt programmet, brukeren ville ikke fortsette
```

Av og til har vi bruk for å lese data fra en fil. Vi åpner og lukker en fil ved hjelp av kommandoene `open` og `close`. Videre kan vi lese linjer fra filen ved kommandoen `readline()`. Denne kan kalles på mange forskjellige måter. Under kalles den på default-måten, da den leser tegn for tegn fra fila adskilt med space, og tolker hvert tegn som numeriske verdier.

```
fid = open('filsomskalleses.txt','r')
c = fid.readline()
fid.close()
```

Tilsvarende kan vi skrive data til en fil ved hjelp av kommandoen `write`.

```
fid = open('filsomskalskrives.txt', 'w');
fid.write('det du skal skrive')
```

Legg merke til hvordan `'r'` er brukt for `readline` og `'w'` er brukt for `write`. Den første angir at fila skal åpnes i lesemodus, den andre angir at fila skal åpnes i skrivemodus. Når man åpner en fil i skrivemodus kan man åpne enten slik at man legger til de nye tegnene til slutt i den eksisterende filen, eller slik at man lager fila på nytt ved å skrive helt over den gamle.

1.11 Strengvariable

I koden over så vi et eksempel på bruk av metoden `input`. Denne returnerer ikke en numerisk verdi, men en *tekststreng*. `svar` er et eksempel på en *strengvariabel*. Ved hjelp av `==` kan vi sammenligne to strenger på samme måte som vi kan sammenligne numeriske verdier.

Oppgaver til seksjon 1.11

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 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 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.*

1.12 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 de utføres når vi skriver

```
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` Når vi skriver

```
run annengradsligning.py
```

utføres alle kommandoene i filen og svarene returneres:

```
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. Variabelene 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:

```
# 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 vaere 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)
```

Igjen returneres svarene $x_1 = -0.5000$ og $x_2 = -1$.

Oppgaver til seksjon 1.12

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 tillikningssystemet*

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

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 liknings-systemet 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"!*

1.13 Anonyme funksjoner og linjefunksjoner

I forrige seksjon så vi hvordan vi kunne lagre funksjoner 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 det finnes derfor to innebygde metoder for å definere funksjoner direkte i kommanduvinduet: *anonyme funksjoner* ("anonymous functions") og *linjefunksjoner* ("one-line functions").

La oss begynne med linjefunksjoner. Hvis vi skriver

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

lagres 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 *
import matplotlib.pyplot as plt

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

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)
```

er variabelene blitt ordnet 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)
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(x,y,z)
plt.show()
```

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)
>>> fig = plt.figure()
>>> ax = fig.gca(projection='3d')
>>> ax.plot_surface(x,y,z)
>>> plt.show()
```

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 IN1900 (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 IN1900 til å beregne dobbeltintegraler.

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

tilnærmer dobbeltintegralet $\int_0^1 \int_{-1}^2 x^2 y dy dx$, med 100 delepunkter langs x- og y-aksen. Svaret blir 0.5, med de fire første desimalene riktig.

Oppgaver til seksjon 1.13

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

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