Oppgave 1.1

Her er det viktig at de passer på å konvertere til riktige enheter, og unngår heltallsdivisjon. Materialtettheten til kuben er 21450.00 kg /m^3, hvilket som betyr at kuben er laget av Platinium.

```
M = 858      # g
V = 40       #cm**3

M_kg = 858*1E-3      # kg
V_m = 40*1E-6        # m**3

mass_density = M_kg/V_m
print "The mass density of the material is: %.2f"%mass_density
```

Oppgave 1.2

Igjen er det viktig de passer på å få riktige enheter. En riktig så skjult 'felle', er at et år må konverteres til sekunder.

```
from math import pi

AU = 1.58*1E-5        # 10^(-5)   lysaar
ly = 9.5*1E15         # 10^(15)   m
G = 6.674*1E-11        # 10^(-11) m^3*kg^(-1)*s^(-2)

AU_km = AU*ly                                  # 10^(10) km

yr = 365*24*60*60                           # 10^(7) s
M_sun = (4*pi**2*(AU_km)**3)/(G*(yr)**2)     # 10^(27)

print "The calculated solar mass is: %g kg"%M_sun
```

Oppgave 1.3

Pass på heltallsdivisjon i `-t/tau`, og at 10 minutter blir konvertert til sekunder. Kanskje noen også kunne funnet på å bruke tallet `e`, istedenfor math modulens `exp`, noe som ikke er feil men heller ikke skal gjøres.

```
from math import exp, log

N0 = 4.5  #kg
t = 10.0*60  #s
# Exercise a)
tau = 1760
N = N0*exp(-t/tau)
print "Remaining weight of carbon-11 is %.4g kg" % N

# Exercise b)
t_half = 1220  #s
tau = t_half/log(2)
N = N0*exp(-t/tau)
print "Remaining weight of carbon-11 is %.4g kg" % N

"""
Remaining weight of carbon-11 is 3.2 kg
Remaining weight of carbon-11 is 3.2 kg
"""
```

```
>>> Remaining weight of carbon-11 is 3.2 kg
```

Oppgave 1.4

Riktig bruk av `math`-modulen og unngå heltallsdivisjon.

```
from math import sqrt,pi,cos

F0 = 1.
x = 1.
n = 4.
m = 3.
v0 = 2.

v = sqrt(v0**2 + 2./m*(cos(x/n) - 1.) )

print "The velocity of the atom at position x = %g is v = %g"%(x,v)
```

Resultat:

```
>>> The velocity of the atom at position x = 1 is v = 1.99481
```

Oppgave 1.5

Linje 1 og 6: Math-modulen sin kvadratrot heter sqrt". Det er eventuelt mulig å navngi importerte funksjoner med følgende syntax: `from math import sqrt as squareroot`

Linje 2 og 3: Selv om det til daglig ofte er vanlig å separere hvert tredje tall med et mellomrom, godtar ikke python syntaxen dette. Det er også mest naturlig å skrive såpas store tall med vitenskapelig notasjon.

Linje 4: For desimaltall i Python må vi bruke et punktum. Komma brukes for å separere elementer.

Linje 6: I Python skrives eksponenter som '**'.

Linje 6, eller 2 og 3: Det forekommer en heltallsdivisjon mellom v^2 og c^2. Vi kan enten initialisere variablene som floats i linje 2 og 3, eller konvertere dem til floats før vi gjør divisjonen i linje 6. Det mest naturlig er det førstenevnte, som er gjort i koden under.

Kravet for å beste oppgaven er å ha fått den til å fungere. Koden skal se ut som noe slikt:

```
from math import sqrt
c = 3e8  #m/s
v = 1e8  #m/s
m = 0.14  #kg

gamma = 1/sqrt(1-(v**2/c**2))
p = m*v*gamma

print p
```

Oppgave 1.6

Hensikten med dette oppgaven er at de skal få se nytten i å lagre verdier som variabler. Det er derfor viktig at passende variabler blir tilordnet passende verdier.

```
me = 9.109e-31          #m
e = 1.602e-19           #C
varepsilon = 8.854e-12  #CV^{-1}m^{-1}
h = 6.626e-34           #Js
c = 3e8                 #m/s

rydberg = me*e**4/(8*varepsilon**2*h**3*c)
print "The Rydberg constant is approximately: %g m^{-1}"%rydberg
```

Resultat:

```
>>> The Rydberg constant is approximately: 1.09617e+07 m^{-1}
```

Oppgave 2.1

Det viktige i denne oppgaven, er at de får kjennskap til hvordan en while loop skal implementeres. Det er viktig at det ikke testes for liket i loopen.

```
v0 = 0
g = 9.81
dt = 0.01
h0 = 10
h1 = 5

time_taken = 0
current_height = h0
while(current_height > h1):
    time_taken += dt
    current_height = v0*time_taken - .5*g*time_taken**2+h0

print "It took the ball approximately %g seconds to pass %g meters"%(time_taken,h1)
```

Resultat:

```
>>> It took the ball approximately 1.01 seconds to pass 5 meters
```

Oppgave 2.2

Hensikten med denne oppgaven er å bli kjent med hvordan man bruker for loops og print formatering til å lage tabeller. Formateringen bør altså helst være ryddig og i rette kolonner, og det skal brukes vitenskapelig notasjon. Det bør helst ikke brukes for mange eller for få desimaler i printen (2-5). Det er helt greit å printe hastigheten enten i meter per sekund og i andeler av lyshastigheten, men bevegelsesmengde bør helst være i $\mathrm{kg\,m/s^2}$.

Pass på at v^2/c^2 i teorien kan gi float division.

```
c = 3e8
m = 1200

# __Exercise a__
print "\n%12s%16s" % ("Vel.(m/s)", "Mom.(kgm/s)")
for i in range(0,10):
    v = 0.1*i*c
    p = m*v
    print "%11.4g%16.4g" % (v,p)

# __Exercise b__
from math import sqrt
print "\n%12s%20s%20s" % ("Vel.(c)", "Clas.Mom.(kgm/s)", "Rel.Mom.(kgm/s)")
for i in range(0,10):
    v = 0.1*i*c
    p_classic = m*v
    gamma = 1/sqrt(1-(v**2/c**2))
    p_rel = m*v*gamma
    print "%10gc%20.4g%20.4g" % (0.1*i, p_classic, p_rel)
```

```
   Vel.(m/s)     Mom.(kgm/s)
           0               0
       3e+07         3.6e+10
       6e+07         7.2e+10
       9e+07        1.08e+11
     1.2e+08        1.44e+11
     1.5e+08         1.8e+11
     1.8e+08        2.16e+11
     2.1e+08        2.52e+11
     2.4e+08        2.88e+11
     2.7e+08        3.24e+11

      Vel.(c)    Clas.Mom.(kgm/s)     Rel.Mom.(kgm/s)
          0c                   0                   0
        0.1c            3.6e+10          3.618e+10
        0.2c            7.2e+10          7.348e+10
        0.3c           1.08e+11          1.132e+11
        0.4c           1.44e+11          1.571e+11
        0.5c            1.8e+11          2.078e+11
        0.6c           2.16e+11            2.7e+11
        0.7c           2.52e+11          3.529e+11
        0.8c           2.88e+11            4.8e+11
        0.9c           3.24e+11          7.433e+11
```

Oppgave 2.3

Dette er ideelle omstendigheter for å bruke en while loop, og bør ikke implementeres med for loop. Pass på at enten første eller siste element i listen må settes utenfor loopen, ettersom vi skal sette en verdi mer enn loopens lengde.

Oppgave c) inkluderer litt komplisert indeksering, med mange mulige løsninger. Noen er selvsagt penere enn andre, men det viktigste er at det fungerer. I koden under ligger det to mulige løsninger. Den første baserer seg på å loope over indekser, og den andre å loope over selve elementene i listene.

```
from math import exp, log

N0 = 4.5
N = N0
t = 0; dt = 60
tau = 1760.0
t_list = [t]; N_list = [N]  # Including initial time and mass in list.

# __Exercise a__
while N > 0.5*N0:
    t += dt
    N = N0*exp(-t/tau)
    N_list.append(N)
    t_list.append(t)
print "%.2f%% is left of the carbon-11 after %d seconds." % (N/N0*100, t)


# __Exercise b__
print "Estimated half-life from simulation = %.2f s" % t_list[-1]
print "Actual half-life from analytical solution = %.2f s" % (tau*log(2))


# __Exercise c__
Nt = []
for i in range(len(t_list)):
    Nt.append([t_list[i], N_list[i]])

print "\n%12s%12s" % ("t [Seconds]","N(t) [Kg]")
for i in range(len(t_list)):
    print "%11d%12.4g" % (Nt[i][0], Nt[i][1])

# __Exercise c, alternative solution__
Nt = []
```

```
for t, N in zip(t_list, N_list):
    Nt.append([t,N])

print "\n%12s%12s" % ("t [Seconds]","N(t) [Kg]")
for Nt_pair in Nt:
    print "%11d%12.4g" % (Nt_pair[0], Nt_pair[1])
```

```
>>> 48.87% is left of the carbon-11 after 1260 seconds.

>>> Estimated half-life from simulation = 1260.00 s
>>> Actual half-life from analytical solution = 1219.94 s
```

t [Seconds]	N(t) [Kg]
0	4.5
60	4.349
120	4.203
180	4.063
240	3.926
300	3.795
360	3.668
420	3.545
480	3.426
540	3.311
600	3.2
660	3.093
720	2.989
780	2.889
840	2.792
900	2.699
960	2.608
1020	2.521
1080	2.436
1140	2.355
1200	2.276
1260	2.199

Oppgave 2.4

Viktig at summeringen blit utført og at de passer på å ikke gjøre helltalsdivisjon.

Her skal det ikke være nødvendig å lagre massene og avstandene i lister. Det hadde vært flott om at det ble tydelig sagt i fra om at lister **ikke er nødvendige** dersom du ser noen gjøre dette.

```
from math import sqrt
G = 6.674*1E-11                #m^3*kg^(-1)*s^(-2)
M = 3.
N = 10

F = 0
for i in xrange(N):
    m_i = i/6. + 2
    r_i = sqrt((i/4.)**2 + 10)
    F += G*(m_i*M)/(r_i)**2

print "The total attractive force on the object with mass M = %g is %g N"%(M,F)
```

Den totale kraften F skal bli:
```
>> The total attractive force on the object with mass M = 3 is 4.65492e-10 N
```

Oppgave 2.5

Her er det bare å sette inn formelen gitt i oppgaven, og iterere over den fra $i = 1$ til $i = 29$. Det er fort gjort å gjøre feil på disse indeksene.

Det er også mulig å gange ut det faktoriserte i-leddet i formelen før man implementerer den, så keep that in mind hvis formelen i koden ser ukjent ut.

Det er også mulig å finne den totale energien ved å summe listen etter loopen.

```
h = 6.626e-34  # Plancks constant, in Js
L = 1e-11  # size of box, in meters
m = 9.11e-31  # mass of particle, in kg

energies = []
total_energy = 0

for i in range(1, 30):  # From E1->E2 to E29->E30
    energy_step = (((i+1)**2-i**2)*h**2)/(8*m*L**2)
    energies.append(energy_step)
    total_energy += energy_step
```

Rettehjelp:
`energies[0] = 1.807e-15`
`energies[-1] = 3.554e-14`
`total_energy = 5.415e-13`

Oppgave 2.6

Her skal de vise at de forstår hvordan de kan oversette en while loop til en for loop. En elementvis for loop som den siste er ikke nødvendig, men gir litt penere kode.

```
from math import sqrt

g = 9.81
r = [2.7,3.43,5.62,7.1]
num_loops = len(r)
v = [0]*num_loops

for i in range(num_loops):
    v[i] = sqrt(r[i]*g)      #in m/s
    v[i] = v[i]*3600/1000.    #convert to km/h

for v_ in v:
    print "Least speed to complete the loop: %.2f km/h"%v_
```

Resultat:

```
>>> Least speed to complete the loop: 18.53 km/h
>>> Least speed to complete the loop: 20.88 km/h
>>> Least speed to complete the loop: 26.73 km/h
>>> Least speed to complete the loop: 30.04 km/h
```

Oppgave 3.1

Alle tre variablene skal sendes inn til funksjonen, ingenting skal være globalt. Heltallsdivisjon kan skje i `-t/tau`.

Test funksjonen bør helst inneholde en `assert` statement, og som nevnt i oppgaven kan ikke toleransen være mindre enn `1e-4` fordi 3.2 kg er en (ganske nøyaktig) tilnærming.

```
from math import exp

def N(N0, tau, t):
    return N0*exp(-t/float(tau))

def test_N():
    tol = 1e-4
    calculated_N = N(4.5, 1760, 600)
    actual_N = 3.2
    assert abs(actual_N - calculated_N) < tol

test_N()
```

Oppgave 3.2

Her skal altså to `n`'er, `L` og `m` tas inn som variabler, og det skal regnes ut differansen mellom to energinivåer.

```
def quantum_energy(n1, n2, L, m):
    h = 6.626e-34
    E1 = (n1**2*h**2)/(8*m*L**2)
    E2 = (n2**2*h**2)/(8*m*L**2)
    return E2 - E1
```

Oppgave 3.3

Fokuset i oppgaven er at de får til å iterere gjennom den innsendte listen av friksjonskoeffsienter, utføre beregninger på dem og lagre resultatene i en liste.

Det er ikke så viktig at `zip` brukes, men fint om de får vite at den finnes og hint til hvordan den kan brukes i denne oppgaven.

```
def x(t,f,v0):
    return v0*t - .5*f*9.81*t**2

def find_distance(v0,dt,frictions):
    g = 9.81
    v = v0
    l = len(frictions)
    lengths = [0]*l
    for i in range(l):
        f = frictions[i]
        T = v0/(f*9.81)
        lengths[i] = x(T,f,v0)
    return lengths


frictions = [0.62,0.3,0.45,0.2]
v0 = 5.

lengths = find_distance(v0,0.0001,frictions)
for l,f in zip(lengths,frictions):
    print "Length: %g with frictioncoefficient: %g "%(l,f)
```

Resultat etter kjøring av løsningsforslaget:

```
>>> Length: 2.05518 with frictioncoefficient: 0.62
>>> Length: 4.24737 with frictioncoefficient: 0.3
>>> Length: 2.83158 with frictioncoefficient: 0.45
>>> Length: 6.37105 with frictioncoefficient: 0.2
```

Oppgave 3.4

Her er det viktig at de får til å lage en if-test i en funksjon (til å beregne poengene) uten å bruke globale variable.

```
from math import cos,pi,sin

def height(t,v0,th):
    g = 9.81
    return -.5*g*t**2 + v0*t*sin(th)

def find_points(h0,h1,y):
    points = 0
    mid_point = (h0 + h1)/2.
    if (h0 <= y <= mid_point):
        points = 1
    elif (mid_point < y <= h1 ):
        points = 2
    return points

h0 = 3
h1 = 3.5
b = 3.5
g = 9.81
th = pi/4

for v0 in [15.,16.,19.,22.]:
    T = b/(v0*cos(th))
    y = height(T,v0,th)
    print "Number of points using v0 = %g: %g"%(v0,find_points(h0,h1,y))
```

Resultat etter kjøring av løsningsforslaget:

```
>>> Number of points using v0 = 15: 0
>>> Number of points using v0 = 16: 1
>>> Number of points using v0 = 19: 1
>>> Number of points using v0 = 22: 2
```

Noen delresultater som kan være til nytte:

- Når $v_0 = 15$, så er $y = 2.9659$
- Når $v_0 = 16$, så er $y = 3.03057617187$
- Når $v_0 = 19$, så er $y = 3.16711218837$
- Når $v_0 = 22$, så er $y = 3.25170971074$

Oppgave 3.5

Poenget med denne oppgaven er å lage to funksjoner som i utgangspunktet skal returnere samme verdi, men fra ulike parametere. Testfunksjonen skal benytte ligningen i oppgaven til å konvertere en gitt tid til posisjon, slik at de to funksjonene kan sammenlignes.

I oppgave c) er det best å sjekke typen til `t` i en if/else block, og løse problemet helt separat.

```
from math import sqrt

# __Exercise a__
def velocity1(t, v0, a=9.81):
    return v0 + a*t

def velocity2(x, v0, a=9.81):
    return sqrt(v0**2 + 2*a*x)

# __Exercise b__
def test_velocity():
    tol = 1e-6
    v0 = 5.5
    t = 3.5
    a = 9.81
    x = v0*t + 0.5*a*t**2

    v1 = velocity1(t, v0)
    v2 = velocity2(x, v0)

    assert (v1 - v2) < tol, "The two functions return different velocities."

# __Exercise c__
def velocity1_improved(t, v0, a=9.81):
    if type(t) is list:
        v = []
        for t_element in t:
            v.append(v0 + a*t_element)
        return v
    else:
        return v0 + a*t
```

Oppgave 3.6

Denne er nok i det tøffere laget fordi det er mye å holde styr på. Her er det meningen at de skal vise hva som er viktig å skille mellom ved for- og while loop, og ved hvilke tilfeller de skal brukes til. Tenker at de viser veldig god beherskelse på pensumet fram til nå om de får til b).

```
# a)
def x(t,f,v0):
    return v0*t - .5*f*9.81*t**2

# b)
def find_distance(v0,dt,frictions):
    g = 9.81
    t = 0
    v = v0

    l = len(frictions)
    lengths = [0]*l
    for i in range(l):

        f = frictions[i]

        while(v > 0):
            t += dt
            v = v0 - f*g*t

        lengths[i] = v0*t - .5*f*g*t**2
        v = v0
        t = 0
    return lengths

# c)
def test_find_distance(lengths,frictions,v0):
    for f,l in zip(frictions,lengths):
        T = v0/(f*9.81)
        expected_length = x(T,f,v0)
        assert abs(expected_length-l)<1E-7 , \
        'Expected: %g, got: %g'%(expected_length,l)


frictions = [0.62,0.3,0.45,0.2]
v0 = 5.

lengths = find_distance(v0,0.0001,frictions)
test_find_distance(lengths,frictions,v0)
```

Delresultat som kan være til nytte:
Listen `lengths` har verdiene `[2.0551774003489998, 4.2473666285, 2.8315777446474995, 6.371049942749998]`

Oppgave 4.1

Viktige punkter er å bruke en try-except-blokk og la programmet kjøre en egendefinert testfunksjon.

```
import sys
from math import pi

# b)
def test_uncertainty(x,p):
    h = 6.626e-34
    assert x*p >= h/(4*pi), \
    "Deltax and deltap does not hold with the uncartainty principle"

# a)
try:
    deltax = float(sys.argv[1])
    deltap = float(sys.argv[2])
    test_uncertainty(deltax,deltap)
except IndexError:
    print "Not enough input arguments"
except ValueError:
    print "Value cannot be converted to float"
```

Delresultater som kan være til nytte:

- $\frac{h}{4\pi} = 5.27280326463 \cdot 10^{-35}$
- $\Delta x_1 \cdot \Delta p_1 = 5.272805 \cdot 10^{-35}$
- $\Delta x_2 \cdot \Delta p_2 = 5.2 \cdot 10^{-35}$

Oppgave 4.2

I denne oppgaven skal det settes opp et input system, som først ser etter verdier i terminalen, og spør etter dem fra brukeren hvis verdiene ikke finnes eller er ugyldige. Disse verdiene brukes så til å regne ut et par enkle funksjoner.

Det er ikke nødvendig å implementere $x(t)$ og $v(t)$ som funksjoner, men det er veldig anbefalt.

I oppgave b) bør det være en `try` statement for å initialisere alle variablene, og to `except` statements for de to mulige feilene som kan oppstå ved denne initialiseringen. Begge spør brukeren om å sette inn verdiene manuelt.

```
def x(t, v0, q, E, m):
    return v0*t + 0.5*q*E/m*t**2

def v(t, v0, q, E, m):
    return v0 + q*E/m*t

m_electron = 9.1e-31
q_electron = -1.6e-19
E = 1e-10

# __Exercise a__
v0 = float(raw_input("Initial velocity of electron: "))
t = float(raw_input("Time at which we wish to study the electron: "))
x_value = x(t, v0, q_electron, E, m_electron)
v_value = v(t, v0, q_electron, E, m_electron)
print "Electron has a position of %.2f m and a velocity of %.2f m/s \
at time %.2f seconds." % (x_value, v_value, t)


# __Exercise b__
import sys
try:
    v0 = float(sys.argv[1])
    t = float(sys.argv[2])
    q = float(sys.argv[3])
    m = float(sys.argv[4])
except IndexError:
    print "Too few command line arguments."
    v0 = float(raw_input("Initial velocity of particle: "))
    t = float(raw_input("Time at which we wish to study the particle: "))
    q = float(raw_input("Charge of particle: "))
    m = float(raw_input("Mass of particle: "))
except ValueError:
    print "Input cannot be converted to float."
    v0 = float(raw_input("Initial velocity of particle: "))
    t = float(raw_input("Time at which we wish to study the particle: "))
    q = float(raw_input("Charge of particle: "))
    m = float(raw_input("Mass of particle: "))

x_value = x(t, v0, q, E, m)
v_value = v(t, v0, q, E, m)
print "Particle has a position of %.2f m and a velocity of %.2f m/s \
at time %.2f seconds." % (x_value, v_value, t)
```

```
>>> Electron has a position of 1321.98 m and a velocity of -43.74 m/s at time 15.00 seconds.
>>> Particle has a position of 3300.00 m and a velocity of 220.00 m/s at time 15.00 seconds.
>>> Particle has a position of 3301.08 m and a velocity of 220.14 m/s at time 15.00 seconds.
```

Oppgave 4.3

Hva som er tilstrekkelig nøyaktig er selvsagt åpent for diskusjon, og ingenting er direkte feil, men en grense ett sted mellom $0.1c$ og $0.5c$ er å foretrekke.

Vi har her valgt å sette $\gamma = 1$ ved det klassiske tilfellet, og bruke samme formel, men det også helt greit å regne de to formlene direkte i if/else-blokken.

De bør helst vende seg til å bruke `raw_input`, og konvertere til float, istedenfor å bruke `input`.

I oppgave b) er det bare å bytte input med terminal argumenter.

I oppgave c) skal de wrappe initialiseringen av variabler i en `try/except` block. Denne bør inneholde hånteringer av både `IndexError` og `ValueError`. I tillegg skal det være to `if/else` tester for å sjekke massen og hastigheten, som skal raise sine egne errors. Alle errors skal ha med forklarende tekst.

```
import sys
from math import sqrt

c = 3e8

# Exercise a) input:
# m = float(raw_input("mass of object: "))
# v = float(raw_input("velocity of object: "))

try:
    m = float(sys.argv[1])
    v = float(sys.argv[2])
except ValueError:
    print "Input cannot be converted to float."
    sys.exit()
except IndexError:
    print "Too few command line arguments."
    sys.exit()

if m < 0:
    raise ValueError, "Mass cannot be negative."
if v > 3e8:
    raise ValueError, "Speed cannot be greater than c."

if v > 1e8:  # Classical case corresponds to gamma = 1.
    gamma = 1/sqrt(1-v**2/c**2)
else:
    gamma = 1

p = m*v*gamma
print "Momentum of object is %g kgm/s" % p
```

Oppgave 4.4

a)

Her er det viktig at de skriver et program som kan lese inn *vilkårlig* antall verdier for massene, helningsvinkelene og friksjonskoeffisientene. Dersom noen glemmer å lukke filene, kan det være flott å si ifra om dette er viktig for å unngå feil i programmet.

b)

Det er fint om de greier å skrive til fil hva hver verdi står for.

```
from math import cos,pi

#a)
with open('slide_books.dat','r') as infile:
    infile.readline()
    masses = [float(value) for value in infile.readline().split()]

    infile.readline()
    degrees = [float(value) for value in infile.readline().split()]

    infile.readline()
    friction_coeff = [float(value) for value in infile.readline().split()]

#b)
g = 9.81
with open('slide_books_result.dat','w') as infile:

    for m in masses:
        infile.write("--- Book with mass %.2f kg ---\n"%m)
        for th in degrees:
            th_rad = (th*pi)/180.
            infile.write("theta = %.2f rad\n"%th_rad)
            for f in friction_coeff:
                infile.write("coefficient of friction = %g\n"%f)
                infile.write("needed friction force is %.2f N\n\n\n"%(f*m*g*cos(th_rad)))
    infile.write("\n")
```

Oppgave 4.5

Det som er nytt her, er `IOError`, men ellers er alt ganske likt som i boka. Det er også viktig å passe på å lukke fila.

```
import sys

try:
    filename = sys.argv[1]
    M = float(sys.argv[2])

    G = 6.674*1e-11
    F = 0
    with open(filename,'r') as infile:
        for line in infile:
            values = line.split()
            m = float(values[0])
            r = float(values[1])
            F += G*m*M/(r**2)

    print "Total force on object with mass %g kg is %g N"%(M,F)

except IndexError:
    print "Error: Not enough arguments. Expected arguments: M filename"
except IOError:
    print "Error: File %s not found"%filename
except ValueError:
    print "Error: Value of M is not valid."
```

Resultat:

```
>>> Total force on object with mass 0.7 kg is 5.85559 N
```

Oppgave 5.1

Her skal listen konverteres til arrays ved bruk av `np.array`. Det skal ikke brukes noen loops for å utføre beregningene.

```
import numpy as np
# a)
N = 10
F = [0]*N
for i in xrange(N):
    F[i] = 30 + 5*i

# b)
F = np.array(F)
F = F/2.
print "Total forces: %g N"%np.sum(F)
```

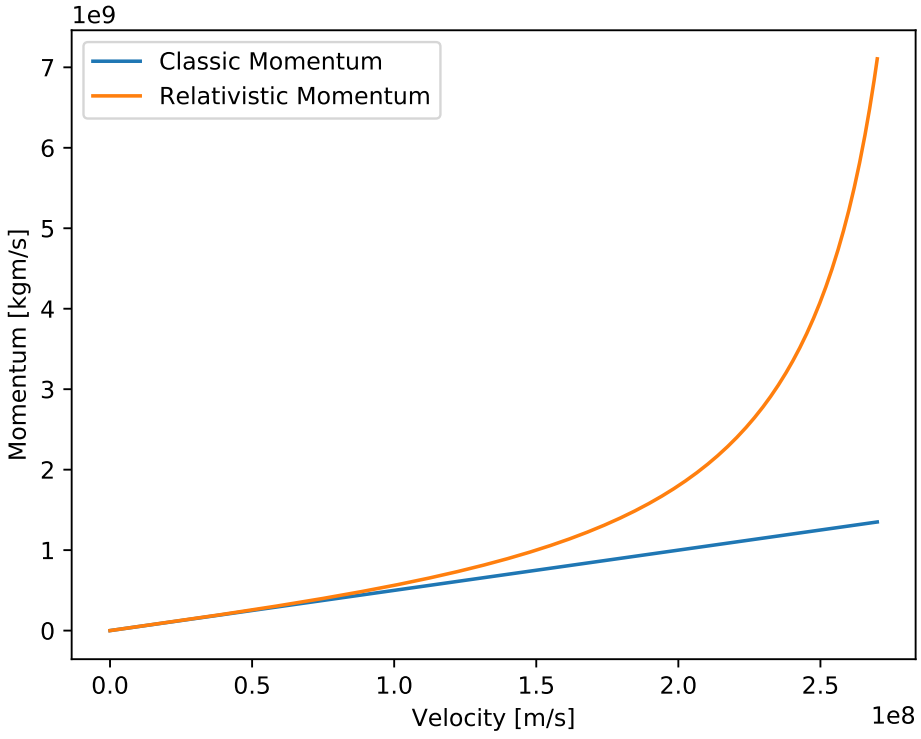
Resultat:

```
>>> Total forces: 262.5 N
```

Oppgave 5.2

I denne oppgaven skal de plotte to funksjoner i samme plott. Bevegelsesmengdene bør helst være funksjoner, og det skal sendes inn en tidsarray i det spesifiserte intervallet.

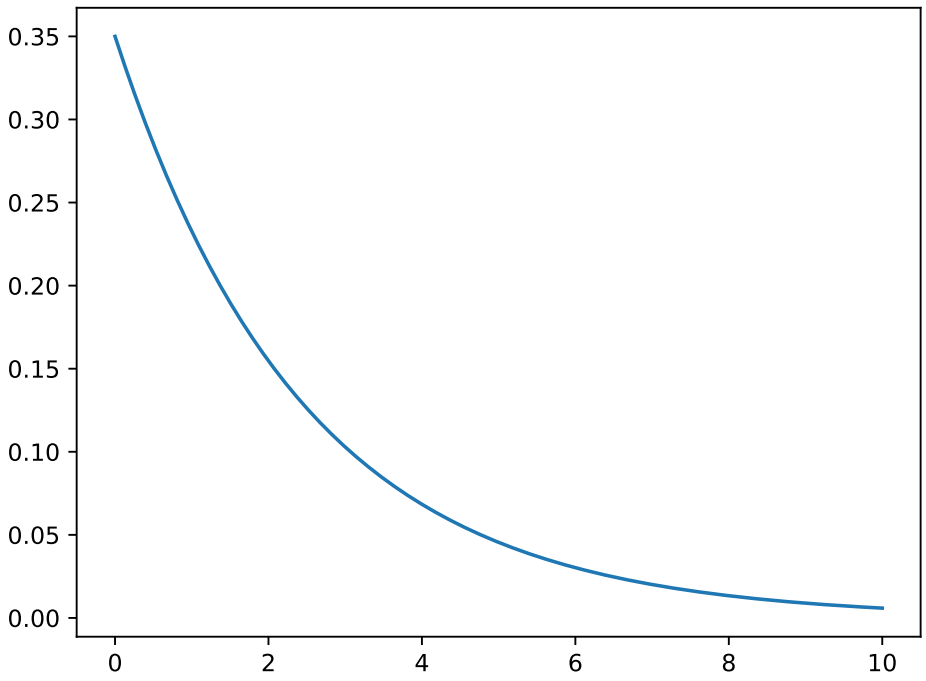
```
import numpy as np
import matplotlib.pyplot as plt

c = 3e8
m = 5.0

def p_clas(v, m):
    return m*v

def p_rel(v, m):
    gamma = 1/(1-(v**2/c**2))
    return m*v*gamma

v = np.linspace(0, 0.9*c, 1001)

plt.plot(v, p_clas(v, m))
plt.plot(v, p_rel(v, m))
plt.legend(["Classic Momentum", "Relativistic Momentum"])
plt.xlabel("Velocity [m/s]")
plt.ylabel("Momentum [kgm/s]")
plt.savefig("fig_momentum_plot.pdf")
```



Oppgave 5.3

Her er det vel egentlig bare å bytte ut listene med arrays, og loopen med et funksjonskall. Det skal ikke en eneste loop eller liste i resultatet.

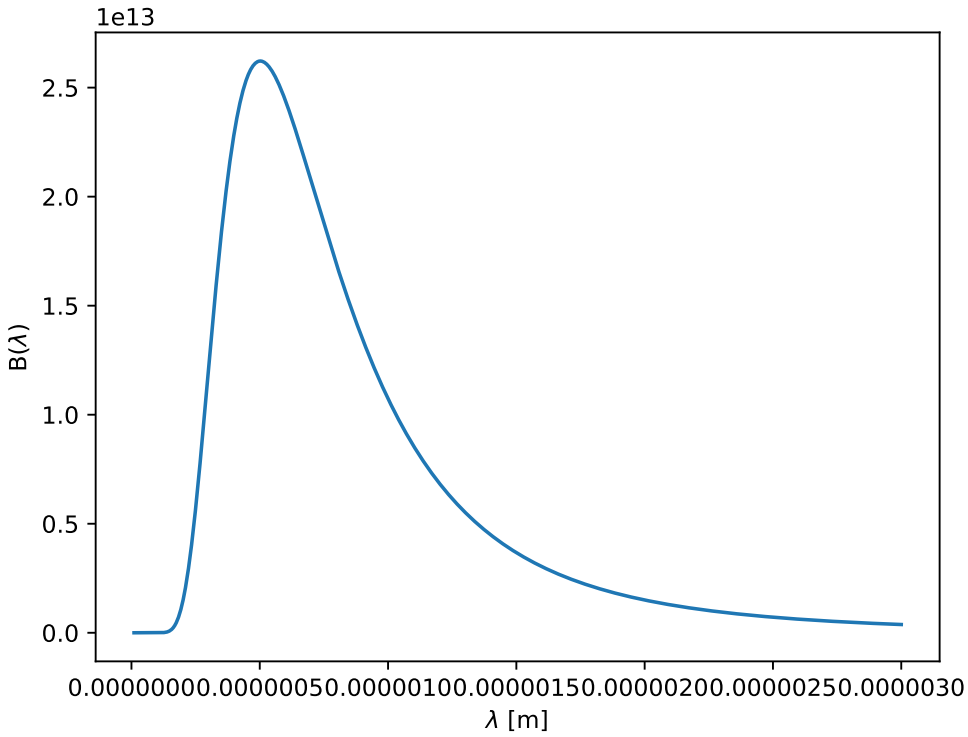
```
import numpy as np
import matplotlib.pyplot as plt

def I(t, R, C, V0):
    return C*V0*np.exp(-t/(R*C))

V0 = 50.0
R = 350.0
C = 0.007

t = 10
n = 1000
dt = float(t)/n

t_array = np.linspace(0, t, n)
I_array = I(t_array, R, C, V0)

plt.plot(t_array, I_array)
plt.savefig('fig_capacitor_vectorization.pdf')
```

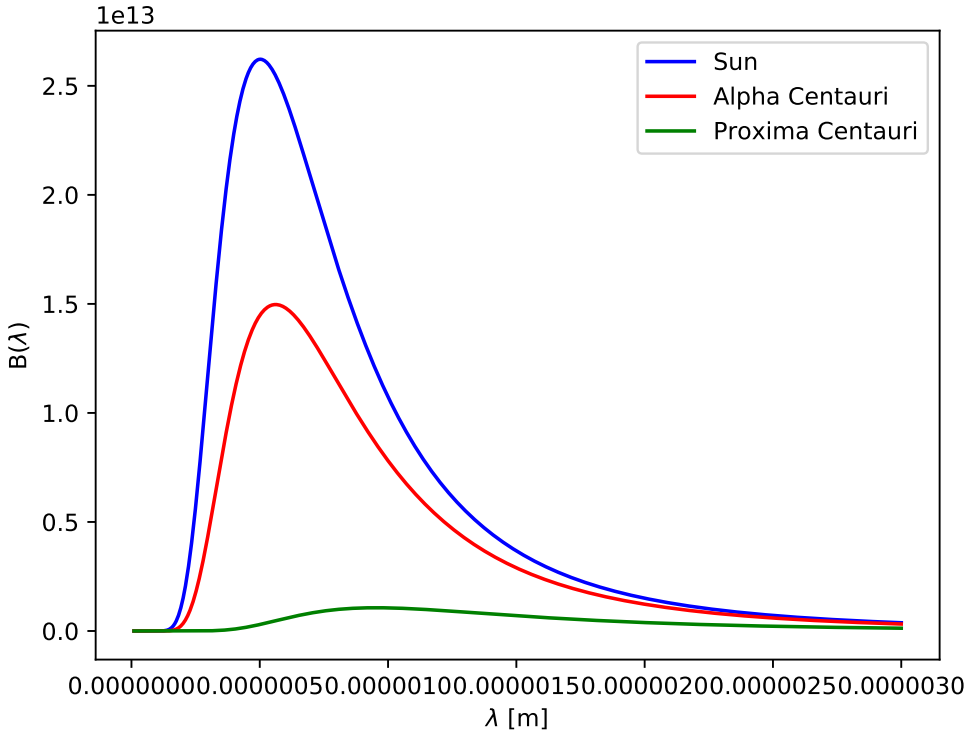


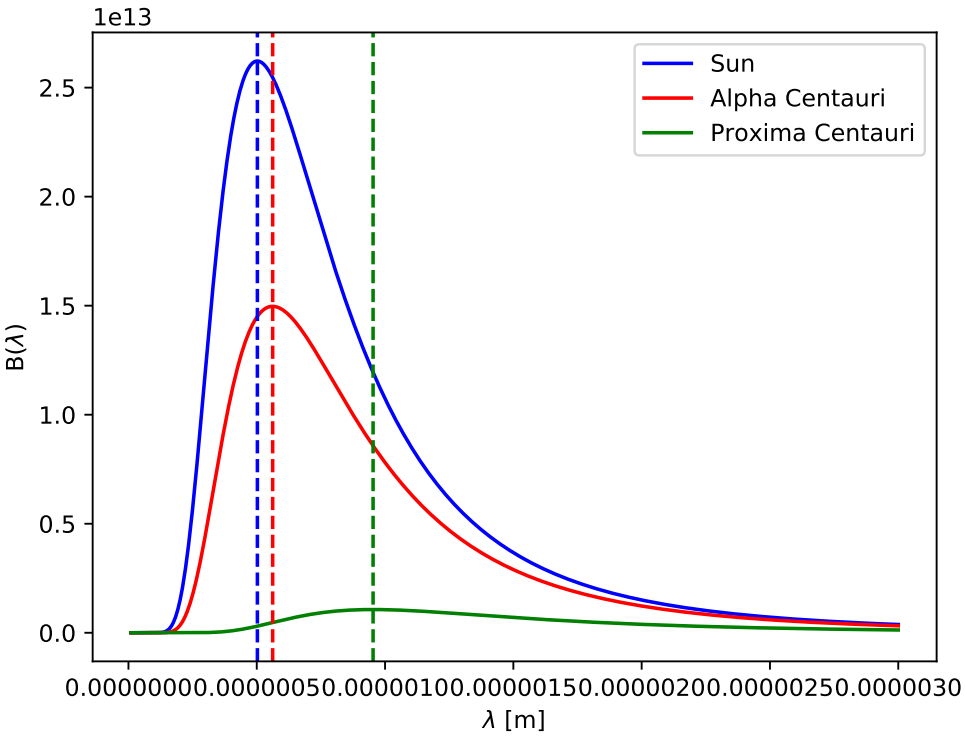
Oppgave 5.4

Viktig å ikke bruke $\lambda = 0$, og at λ starter på 10 nm som spesifisert i oppgaven, for å unngå å dele på 0. Det bør ideelt være satt av nok punkter til λ-arrayet til at grafen ikke ser hakkete ut (type 50+). Det er fint med enheter på aksene, men $B(\lambda)$ har en unødvendig komplisert enhet, så det er ikke så farlig om den ikke er med.

```
import numpy as np
import matplotlib.pyplot as plt

h = 6.62e-34
k = 1.38e-23
c = 3e8
wavelengths = np.linspace(1e-8, 3e-6, 1001)

def B(lamb, T):
    return 2*h*c**2/lamb**5*1/(np.exp(h*c/(lamb*k*T))-1)

# __Exercise a__
T_Sun = 5772.0
plt.plot(wavelengths, B(wavelengths, T_Sun))
plt.xlabel("$\lambda$ [m]")
plt.ylabel("B($\lambda$)")
plt.savefig("fig_Planck_curves_a.pdf")
plt.clf()

# __Exercise b__
T_Alpha = 5160.0
T_Proxima = 3042.0
plt.plot(wavelengths, B(wavelengths, T_Sun), color = "b", label = "Sun")
plt.plot(wavelengths, B(wavelengths, T_Alpha), color = "r", label = "Alpha Centauri")
plt.plot(wavelengths, B(wavelengths, T_Proxima), color = "g", label = "Proxima Centauri")
plt.xlabel("$\lambda$ [m]")
plt.ylabel("B($\lambda$)")
plt.legend()
plt.savefig("fig_Planck_curves_b.pdf")
plt.clf()

# __Exercise c__
b = 2.9e-3
lamb_max_Sun = b/T_Sun
lamb_max_Alpha = b/T_Alpha
lamb_max_Proxima = b/T_Proxima

plt.plot(wavelengths, B(wavelengths, T_Sun), color = "b", label = "Sun")
plt.plot(wavelengths, B(wavelengths, T_Alpha), color = "r", label = "Alpha Centauri")
plt.plot(wavelengths, B(wavelengths, T_Proxima), color = "g", label = "Proxima Centauri")
plt.xlabel("$\lambda$ [m]")
plt.ylabel("B($\lambda$)")
plt.axvline(x=lamb_max_Sun, ls="--", color = "b")
plt.axvline(x=lamb_max_Alpha, ls="--", color = "r")
plt.axvline(x=lamb_max_Proxima, ls="--", color = "g")
plt.legend()
plt.savefig("fig_Planck_curves_c.pdf")
plt.clf()
```


1e13
2.5
2.0
1.5
1.0
0.5
0.0
B(λ)
0.00000000.00000050.00000100.00000150.00000200.00000250.0000030
λ [m]


1e13
Sun
Alpha Centauri
Proxima Centauri
2.5
2.0
1.5
1.0
0.5
0.0
B(λ)
0.00000000.00000050.00000100.00000150.00000200.00000250.0000030
λ [m]


1e13
2.5
2.0
1.5
1.0
0.5
0.0
B(λ)
0.00000000.00000050.00000100.00000150.00000200.00000250.0000030
λ [m]
Sun
Alpha Centauri
Proxima Centauri

Oppgave 5.5

$y(t)$ bør implementeres som en funksjon.

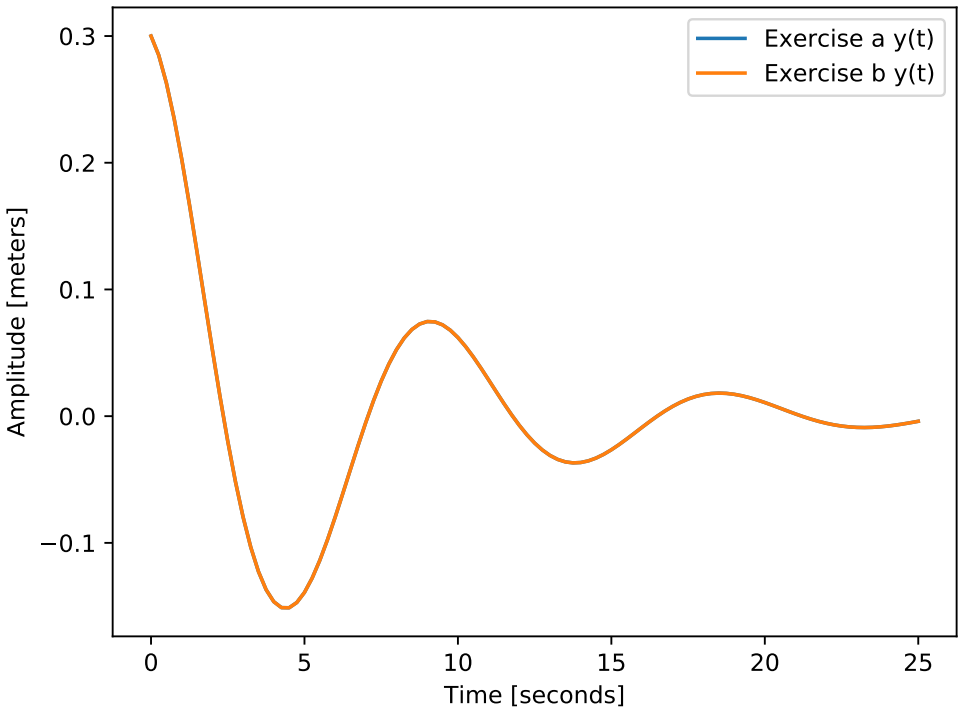
Pass på integer division i k/m.

Akkurat hvordan for loopen i a) er konstruert er ikke så viktig så lenge den funker.

I b) skal koden være vektorisert, og alle loops og lister skal være borte.

a) og b) skal gi helt identiske arrays, som betyr at plottene skal overlappe 100%.

```
import matplotlib.pyplot as plt
from numpy import exp, cos, sqrt, linspace, zeros

def y(t, A, gamma, k, m):
    return A*exp(-gamma*t)*cos(sqrt(k/m)*t)

A = 0.3  # m
gamma = 0.15  # s^-1
k = 4.0  # km/s^2
m = 9.0  # kg

# __Exercise a__
length = 101  # length of arrays
time_interval = 25.0  # time interval we are studying
step_size = time_interval/(length - 1)  # time between two array elements

t_array = zeros(101)
y_array = zeros(101)
y_array[0] = A  # Initial position at t = 0

for i in range(length-1):
    t_array[i+1] = t_array[i] + step_size
    y_array[i+1] = y(t_array[i+1], A, gamma, k, m)


# __Exercise b__
t_array2 = linspace(0, time_interval, length)
y_array2 = y(t_array2, A, gamma, k, m)

plt.plot(t_array, y_array)
plt.plot(t_array2, y_array2)
plt.legend(["Exercise a y(t)", "Exercise b y(t)"])
plt.xlabel("Time [seconds]")
plt.ylabel("Amplitude [meters]")
plt.savefig("fig_oscilating_spring.pdf")
```

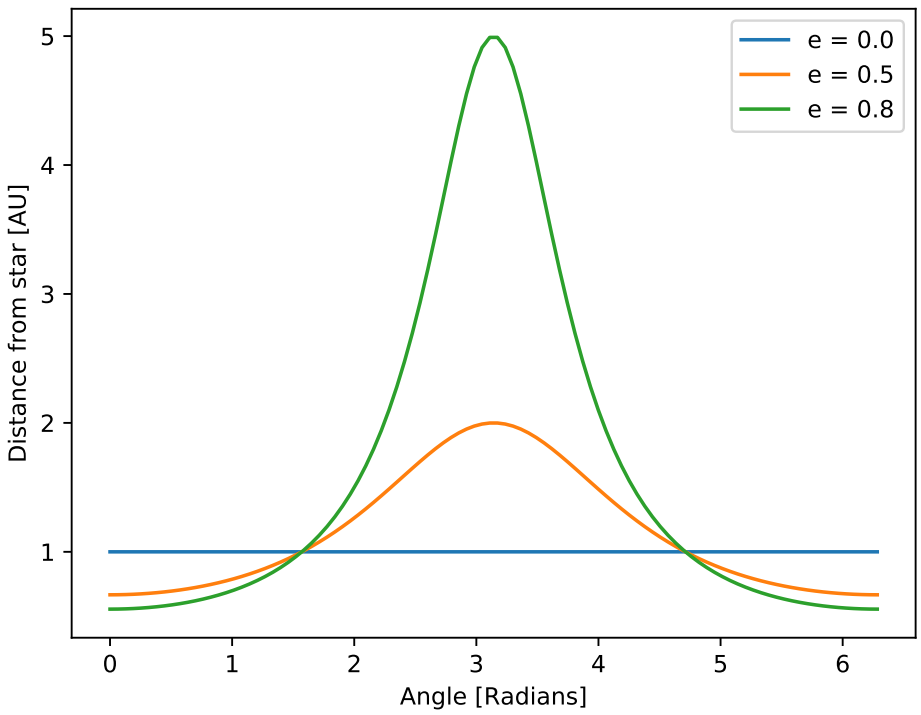

Exercise a y(t)
Exercise b y(t)
0.3
0.2
0.1
0.0
−0.1
Amplitude [meters]
0
5
10
15
20
25
Time [seconds]

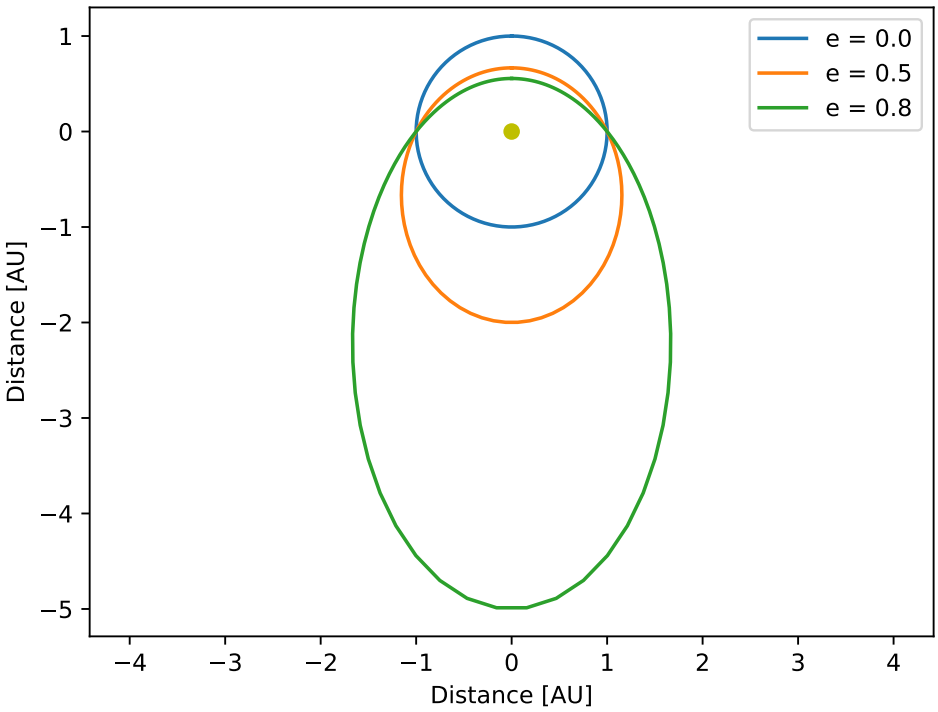
Oppgave 5.6

Det er vel egentlig bare å plotte funksjonen oppgitt i oppgaven med de gitte parameterene mot vinkelintervallet.

I b) skal arrayene fra oppgave a) settes i formelen som konverterer polarkoordinater til kartesiske koordinater, og x og y skal plottes mot hverandre.

```
import numpy as np
import matplotlib.pyplot as plt

def r(theta, e, p):
    return p/(1+e*np.cos(theta))

# __Exercise a__
p = 1.0
theta = np.linspace(0, 2*np.pi, 100)
for e in [0, 0.5, 0.8]:
    radius = r(theta, e, p)
    plt.plot(theta, radius, label = 'e = %.1f' % e)
plt.xlabel('Angle [Radians]')
plt.ylabel('Distance from star [AU]')
plt.legend()
plt.savefig('fig_planetary_motion1.pdf')
plt.clf()

# __Exercse b__
p = 1.0
theta = np.linspace(0, 2*np.pi, 100)
for e in [0, 0.5, 0.8]:
    radius = r(theta, e, p)
    x = radius*np.sin(theta)
    y = radius*np.cos(theta)
    plt.plot(x, y, label ='e = %.1f' % e)
plt.plot(0, 0, 'yo')
plt.xlabel('Distance [AU]')
plt.ylabel('Distance [AU]')
plt.axis('equal')
plt.legend()
plt.savefig('fig_planetary_motion2.pdf')
plt.clf()
```



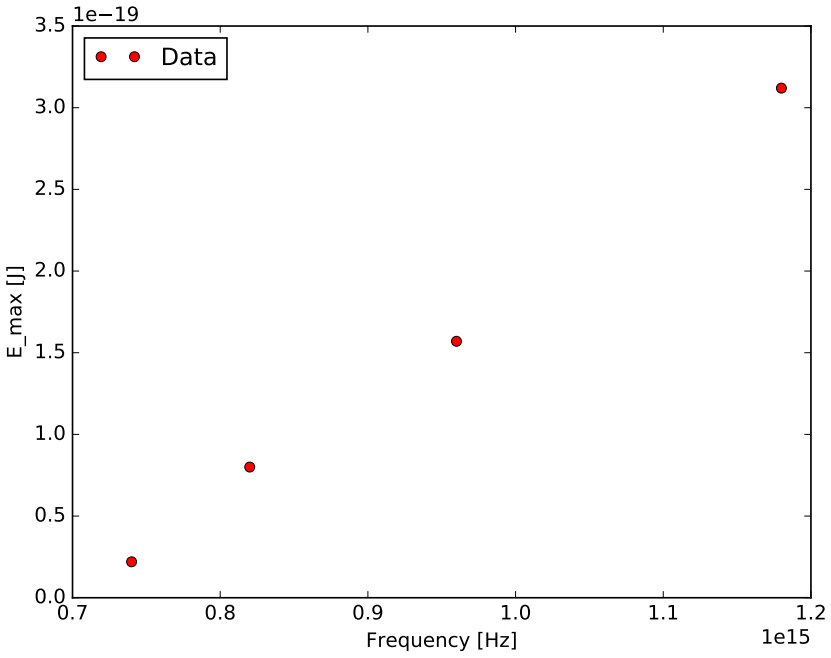

e = 0.0
e = 0.5
e = 0.8
Distance [AU]
Distance [AU]
1
0
−1
−2
−3
−4
−5
−4
−3
−2
−1
0
1
2
3
4

Oppgave 5.7

I denne oppgaven brukes `np.polyfit(x,y,1)` - en 'ny' funksjon som utfører en regresjon mellom de gitte punktene langs hhv. x- og y-aksen. Tallet `1` forteller bare at vi tilpasser et førstegradspolynom til punktene.

```
import matplotlib.pyplot as plt
import numpy as np

# a)
f = np.array([ 1.18,0.96,0.82,0.74])*1e15
E_max = np.array([ 3.12,1.57,0.8,0.22])*1e-19

plt.figure()
plt.plot(f,E_max,'ro')
plt.legend(['Data'],loc = 'upper left')
plt.xlabel('Frequency [Hz]')
plt.ylabel('E_max [J]')


# b)
h,w = np.polyfit(f,E_max,1)
print "Estimate of h: %g"%h

# c)
plt.figure()
plt.plot(f,E_max,'ro',f,h*f + w,'b')
plt.legend(['Data','Estimated line'],loc = 'upper left')

plt.show()
```

a)

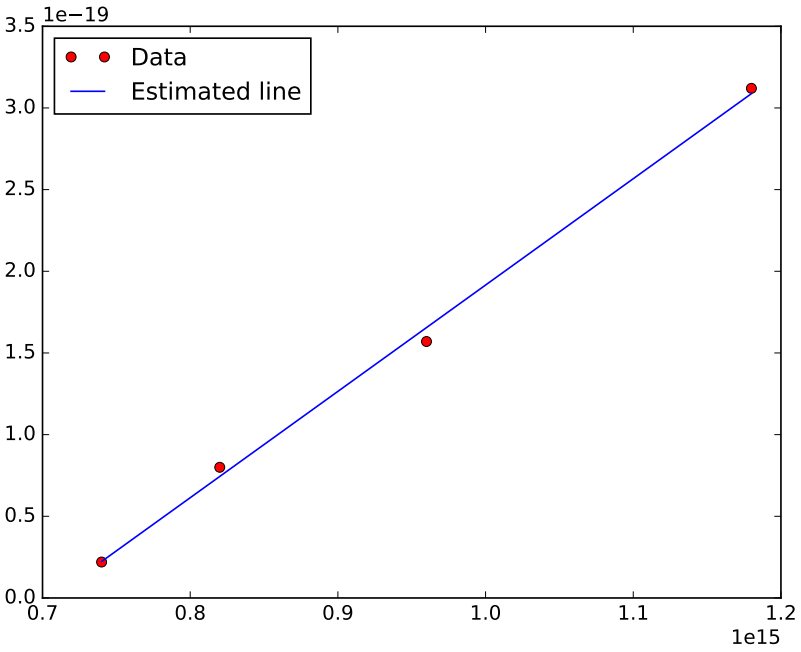


b)

Estimert h:

```
>>> Estimate of h: 6.50987e-34
```

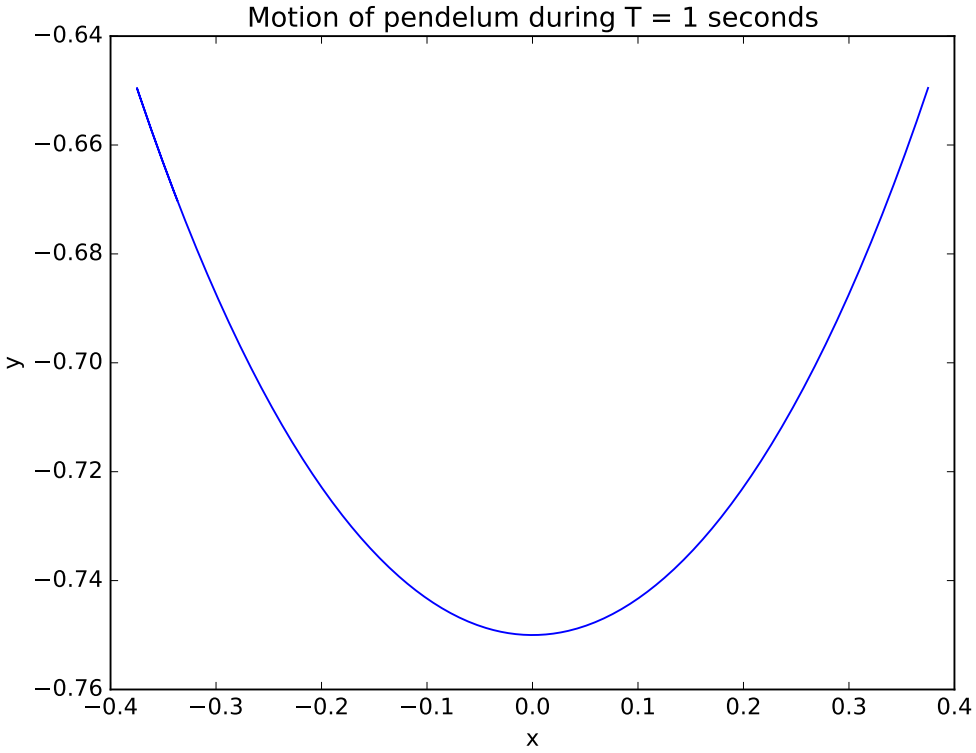
c)



Oppgave 5.8

Greit å bruke arrays her.

```
import numpy as np
import matplotlib.pyplot as plt

R = 0.75          # m

g = 9.81
omega = np.sqrt(g/R)
ph0 = np.pi/6

T = 1
t = np.linspace(0,T,1000)
th = ph0*np.cos(omega*t)

x = R*np.sin(th)
y = -R*np.cos(th)

plt.figure()
plt.plot(x,y)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Motion of pendelum during T = 1 seconds')
plt.savefig('fig_pendelum.pdf')
plt.show()
```


Motion of pendelum during T = 1 seconds
−0.64
−0.66
−0.68
−0.70
−0.72
−0.74
−0.76
y
−0.4
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4
x

Oppgave 5.9

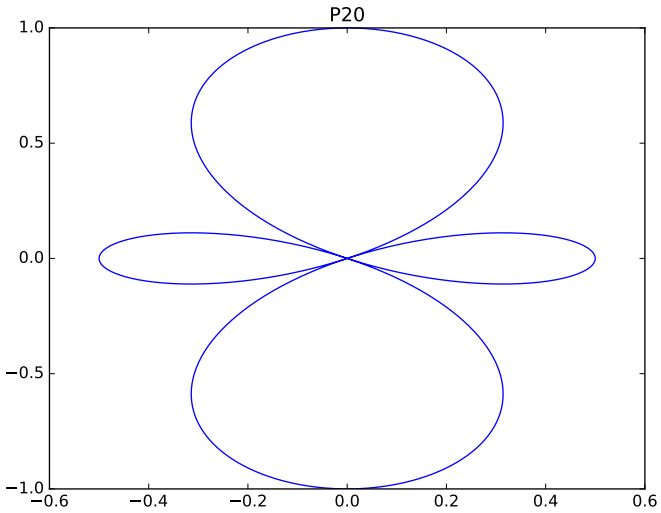
Her er det et poeng i at de skal plotte alle baner i *ett* plott, samt kunne bruke `plt.legends`. Det burde komme tydelig fram hva som skiller banene fra hverandre.

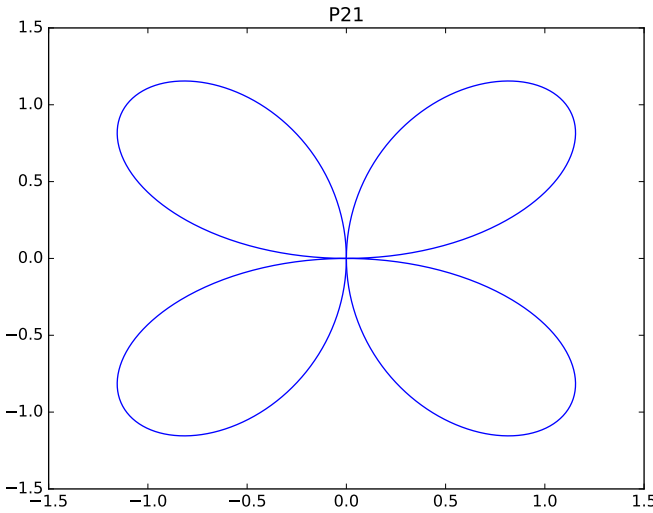
```
import numpy as np
import matplotlib.pyplot as plt

g = 9.81
v0 = 16
for alpha in [np.pi/6,np.pi/4,np.pi/3]:
    t = np.linspace(0,3.5/(v0*np.cos(alpha)))
    y = -.5*g*t**2 + v0*t*np.sin(alpha)
    x = v0*t*np.cos(alpha)
    plt.plot(x,y)

plt.legend(['pi/6','pi/4','pi/3'])
plt.show()
```

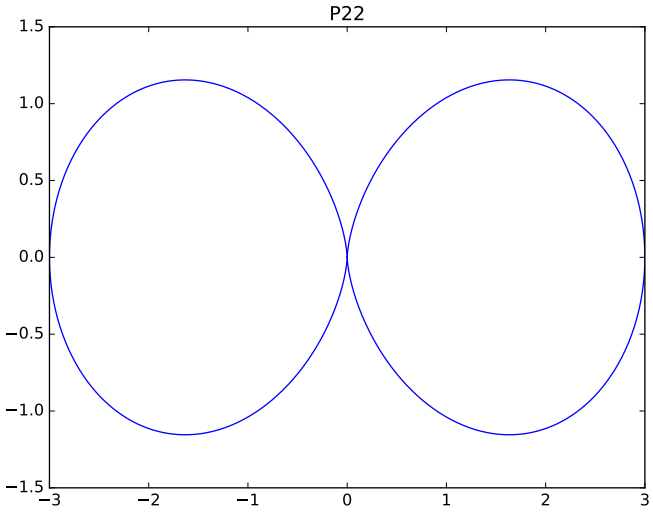
Oppgave 5.10

Mye likt fra boken. Det som kan være utfordrende, er å beholde to grafer i ett plott. Dette kan gjøres ved å enten starte en ny figur med `np.figure()` (fortrukket), er ha `np.show()` i selve loopen (greit, men ikke så elegant).

```
import matplotlib.pyplot as plt
import numpy as np

def P20(th):
    return .5*(3*np.cos(th)**2 - 1)
def P21(th):
    return 3*np.sin(th)*np.cos(th)
def P22(th):
    return 3*np.sin(th)**2

th = np.linspace(0,np.pi,1000)

for function in [P20,P21,P22]:
    r = np.abs(function(th))
    a = r*np.sin(th)
    b = r*np.cos(th)

    plt.figure()
    plt.plot(a ,b,'b',-a,b,'b')

plt.show()
```







Oppgave 6.1

Filen må vellykket være lest, og all informasjonen må være samlet i tre dictionaries med planet-navnene som keys. Om noen mot formodning skulle klare å regne massetetthet feil så er ikke det så viktig.

Til oppgave c) er det selvsagt foretrukket om det brukes printf formatering til å få all dataen i rette kolonner, men det er godkjent så lenge det er minst et space mellom dataen på hver linje (ettersom det da er lesbart av et program).

```
# __Exercise a__
with open("solar_system_data.dat", "r") as infile:
    infile.readline(); infile.readline()  # Skipping two first lines.
    distance = {}
    mass = {}
    radius = {}
    for line in infile:
        words = line.split()
        distance[words[0]] = float(words[1])
        mass[words[0]] = float(words[2])
        radius[words[0]] = float(words[3])

# __Exercise b__
from math import pi
density = {}
for planet in mass:  # Looping over a dictionary gives its keys.
    planet_volume = 4.0/3*pi*(radius[planet]*1000)**3
    density[planet] = mass[planet]/planet_volume

# __Exercise c__
with open("solar_system_data2.dat", "w") as outfile:
    outfile.write("%8s %16s %16s %16s %16s\n" % ("Object", "Distances[km]", "Mass[kg]", "
        Radius[km]", "Densities[kg/m^3]"))
    for planet in mass:
        outfile.write("%8s %16.2e %16.2e %16d %16d\n" % (planet, distance[planet], mass[
            planet], radius[planet], density[planet]))

# __Exercise d__
solar_system = {}
for planet in mass:
    solar_system[planet] = {}
    solar_system[planet]["mass"] = mass[planet]
    solar_system[planet]["distance"] = distance[planet]
    solar_system[planet]["radius"] = radius[planet]
    solar_system[planet]["density"] = density[planet]

number_of_earths = solar_system["Pluto"]["distance"]/solar_system["Earth"]["radius"]
print "Number of earths from Sun to Pluto = %.2f" % number_of_earths
```

```
>>> Number of earths from Sun to Pluto = 925054.88
```

Oppgave 6.2

a)

Viktig at symbolene til konstantene hentes ut og brukes som nøkkel til tilhørende verdier. For å hente ut nødvendig informasjon, er ganske `split()` greiest å bruke. Siden navnene på konstantene er av vilkårlig lengde, må programmet hente ut verdier bakfra i listen etter `split()` har blitt kalt.

b)

Poenget er nå at de skal bruke dictionary-et til å hente ut de nødvendige verdiene for å utføre beregningen beskrevet i oppgaven. Fint om de er påpasselige på hvilken enhet verdien de skriver ut er i.

```
#a)
d = {}
with open('physics_constants.dat','r') as infile:
    for line in infile:
        data = line.split()
        name = data[-3]
        value = float(data[-2])
        d[name] = value

#b)
ke = d['ke']
me = d['me']
e = d['e']
hbar = d['hbar']

factor = (ke**2*me*e**4)/(2*hbar**2)
print "In J: %g\nIn eV: %g"%(factor,factor/e)
```

Resultat:

```
>>> In V: 2.17987e-18
    In eV: 13.6057
```

Oppgave 6.3

Her er det viktig at de behersker `split()` for å få de nødvendige verdiene i lister. Det er også nyttig i oppgaven å dele stringsene etter et gitt argument til `split` istedenfor mellomrom.

```
# Her er det viktig at de faar til split riktig

# a)
with open('friction_coefficients_data.dat','r') as infile:
    materials = infile.readline().split()
    values = infile.readline().split()

# b)
m = 2.5
g = 9.81
print "Material of object | Material of surface | Dynamic friction"
for i,pair in enumerate(materials):
    mat = pair.split('-')
    N = m*g*float(values[i])
    print "%10s  %20s  %20g "%(mat[0],mat[1],N)
```

Resultat:

```
Material of object | Material of surface | Dynamic friction
steel                 steel                14.715
steel                 ice                   1.22625
ice                   ice                   0.73575
ice                  rubber                 0.4905
wood                  wood                  7.3575
nickel                glass                19.1295
wood                  stone                 6.13125
steel                 plexiglass           11.0363
diamond               metal                 2.4525
```

Oppgave 6.4[1]

a)

Viktig å hente ut riktig bynavn, og passe på at programmet tar hensyn til navn med eller uten mellomrom.

b)

Her er det viktig å hente ut riktig bynavn, og hente ut g for hver tilhørende by med dictionaryet fra a). Utfordringen blir nok å vite hvordan testene skal foregå.

```
# a)
gs = {}
with open('data_different_g.dat','r') as infile:
    line = infile.readline().split()

    while(line[0] == 'City:'):

        # Test if the name contains a space
        if (len(line) > 3):
            city = line[1] + " "+ line[2]
        else:
            city = line[1]

        g = line[-1][2:]
        gs[city] = float(g)
        line = infile.readline().split()

# b)
with open('on_Earth.dat','r') as infile:
    line = infile.readline().split()

    while(line[0] != 'On'):

        start = 2
        city1 = line[0]
        if(line[1] != 'to'):
            city1 += " "+line[1]
            start += 1

        city2 = ' '.join(line[start:])

        print str(gs[city1]) + " to " + str(gs[city2])
        line = infile.readline().split()
```

[1] Referanse til Samael - On Earth

Oppgave 7.1

Husk at 'self' parameteren må være med i alle metoder, selv om de ikke tar noen argumenter.

Jeg kan forestille meg at det er mange måter å mislykkes i å tildele en attribute til en klasse, men enten funker det eller ikke. Det er selvsagt ikke lov å endre klasse-koden til å ha en ny attribute...

```
from math import pi

# __Exercise a__
class Planet:
    def __init__(self, name, radius, mass):
        self.name = name
        self.radius = radius
        self.mass = mass

    def density(self):
        return mass / 4/3.0*pi*r**3

    def print_info(self):
        print "The planet %s has a radius %.2f, a mass %.2f, and a density %.2f" % \
        (self.name, self.radius, self.mass, self.density())

# __Exercise b__
planet1 = Planet("Earth", 10, 10)
planet1.population = 7497486172
print planet1.name, "has a population of", planet1.population
```

Oppgave 7.2

a)

Her er det viktig at de får definert funksjonen som beskrevet i oppgaven ordentlig. Det er viktig de sender inn en partikkelinstanse som parameter for å regne ut Coulumbs lov for at studentene skal lære seg å hente ut attributter. For å finne avstanden mellom to partikler har `np.linalg.norm` blitt foreslått ,men andre metoder som finner den euklidske normen mellom posisjonene er også helt greit.

b)

En helt standard testfunksjon der partikkelinstanser opprettes og testes.

```
import numpy as np

# a)
class Particle:

    def __init__(self,position,q):
        self.position = position
        self.q = q

    def interact(self,other_particle):
        pos = self.position
        pos_other = other_particle.position
        r = np.linalg.norm(pos-pos_other)

        q = self.q
        q_other = other_particle.q

        ke = 8.998*1e9        # N m^2/C^2

        return abs(ke*q*q_other/r**2)

# b)
def test_interaction():
    p1 = Particle(np.array([0,0]),-1.602*1e-19)
    p2 = Particle(np.array([0,30*1e-3]),1.602*1e-19)
    F = p1.interact(p2)
    expected = 2.565833688*1e-15
    assert abs(F-expected)<1e-14,'Expected: %g, got: %g'%(expected,F)

if __name__ == '__main__':
    test_interaction()
```

Oppgave 7.3

Dekomponeringen gir at
$x(t) = x_0 + v_{x0}t$
$y(t) = y_0 + v_{y0}t + 0.5at^2$
$v_x(t) = v_{x0}$
$v_y(t) = v_{y0} + at$

Testfunksjonen skal bare velge en t-verdi, sammenligne returen fra de innebyggede metodene med beregnede verdier.

Energi testfunksjonen velger to t-verdier, og sammenligner resultatet av summen av de to oppgitte energifunksjonene.

```
from math import sqrt

# __Exercise a__
class ObjectMovement:
    def __init__(self, x0, y0, vx0, vy0, a=-9.81):
        self.x0, self.y0, self.vx0, self.vy0, self.a = x0, y0, vx0, vy0, a

    def position(self, t):
        x = self.x0 + self.vx0*t
        y = self.y0 + self.vy0*t + 0.5*self.a*t**2
        return x, y

    def velocity(self, t):
        vx = self.vx0
        vy = self.vy0 + self.a*t
        return vx, vy

def test_pos_vel():
    x0 = 10; y0 = 20; vx0 = 5; vy0 = 30; a = -9.81; t = 3; tol = 1e-6
    ball = ObjectMovement(x0,y0,vx0,vy0,a)

    x1_calc = 25; y1_calc = 65.855
    vx1_calc = 5; vy1_calc = 0.57

    x1, y1 = ball.Position(t)
    vx1, vy1 = ball.Velocity(t)

    assert abs(x1 - x1_calc) < tol and abs(y1 - y1_calc) < tol,\
    "Position doesn't match"
    assert abs(vx1 - vx1_calc) < tol and abs(vy1 - vy1_calc) < tol,\
    "Velocity doesn't match"

# __Exercise b__
def test_energy_conservation():
    x0 = 10; y0 = 20; vx0 = 5; vy0 = 30; a = -9.81; t = 3; tol = 1e-6
    ball = ObjectMovement(x0,y0,vx0,vy0,a)
    v0 = sqrt(vx0**2 + vy0**2)
    Ek0 = 0.5*v0**2
    Ep0 = - a*y0

    x1, y1 = ball.Position(t)
    vx1, vy1 = ball.Velocity(t)
    v1 = sqrt(vx1**2 + vy1**2)
    Ek1 = 0.5*v1**2
    Ep1 = - a*y1

    assert abs((Ek0+Ep0)-(Ek1+Ep1)) < tol, "Energy not conserved"
```

Oppgave 7.4

Det nye er å implementere og bruke en `__str__`-funksjon. Viktig at de passer på at helltallsdivisjon ikke oppstår når de bruker instanser av klassen.

```
from math import sin,sqrt,pi # Eventually import from numpy
class Runner:
    #a)
    def __init__(self,mass,v0,alpha):
        self.mass = float(mass)
        self.v0 = float(v0)
        self.alpha = float(alpha)

    #b)
    def __str__(self):
        return "Sprinter with \n mass: %g kg \n initial velocity: %g m/s\n angle: %g
            degrees"\
        %(self.mass,self.v0,self.alpha)

    #c)
    def finish_runtime(self,dist):
        g = 9.81
        F_ = 400

        m = self.mass
        a = self.alpha
        v0 = self.v0

        # Remember to convert to radians
        p_ = g*sin(a*pi/180) + (1./m)*F_

        return -v0/p_ + sqrt(v0**2 + 2*dist*p_)/p_

if __name__ == '__main__':
    r1 = Runner(80,5,30)
    print r1

    dist = 100;
    print "Finish runtime for distance %g m: %g s"%(dist,r1.finish_runtime(dist))
```

Resultat etter kjøring av denne koden:

```
>>> Sprinter with
      mass: 80 kg
      initial velocity: 5 m/s
      angle: 30 degrees
>>> Finish runtime for distance 100 m: 4.017 s
```

Oppgave 7.5

Her er hovedfokuset i å bruke instanser og utføre beregner på dem utenfor klassen.

```
import numpy as np
class Particle:

    def __init__(self,position,mass):
        self.position = position
        self.mass = mass

if __name__ == '__main__':
    particles  = []
    for j in range(5):
        particles.append(Particle(np.array([j,2*j]),j*1e-30))

    total_mass = 0
    com = 0
    for particle in particles:
        total_mass += particle.mass
        com += particle.mass*particle.position
    com /= total_mass

    print "Center of mass in this system is:"
    print com
```

Resultat:

```
>>> Center of mass in this system is:
[ 3.  6.]
```

Oppgave 8.1

Her er poenget at de skal kunne generere tilfeldige verdier som er like sannsynlige å få, og bruke disse. Det er helt greit om de ikke bruker vektoriserng for å teste, men det hadde vært fint om de ble fortalt at dette finnes.

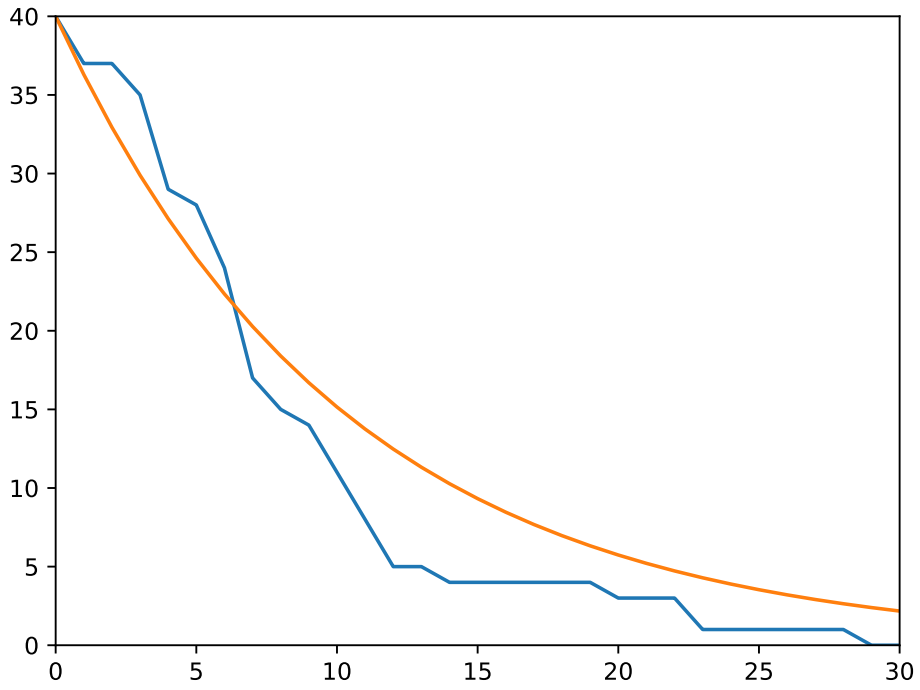
```
import numpy as np
import matplotlib.pyplot as plt
def y(t,v0,m):
    g = 9.81
    return -g*.5*t**2 + v0*t
def v(t,v0,m):
    g = 9.81
    return -g*t + v0

def test_for_conservation(N,v0,m):
    g = 9.81
    T_max = v0/g
    N = 1000
    t1 = np.random.uniform(0,T_max,N)
    t2 =  np.random.uniform(0,T_max,N)
    energies_total1 = m*g*y(t1,v0,m)+.5*m*v(t1,v0,m)**2
    energies_total2 = m*g*y(t2,v0,m)+.5*m*v(t2,v0,m)**2
    success =  np.abs( energies_total2 - energies_total1) < 1e-10
    assert np.sum(success) > 0, "energy not conserved!"

m = 0.057
v0 = 17.
test_for_conservation(1000,v0,m)
```

Oppgave 8.2

Det viktige her er at de på en eller annen måte har en array av 0'er og 1'ere, som de looper over og flipper med en random test. Det totale antallet 1'ere ved hvert tidssteg skal så lagres i en ny array. Det er ikke lov å på en eller annen måte basere seg på den analytiske formelen.

Et typisk plott for $N = 40$ atomer er vist under, men merk at dette kan se ganske annerledes ut pga tilfeldigheter. Plottet skal konvergere godt mot den analytiske løsningen for store N (type $N > 1000$).

```
import numpy as np
import random
import matplotlib.pyplot as plt

# __Exercise a+b__
number_of_atoms = 40
seconds = 30
p = 0.0926

atoms_array = np.ones(number_of_atoms)
atoms_over_time = np.zeros(seconds+1)
atoms_over_time[0] = number_of_atoms

for t in range(seconds):
    for i in range(number_of_atoms):
        if atoms_array[i] == 1:
            if random.random() < p:
                atoms_array[i] = 0
    atoms_over_time[t+1] = np.sum(atoms_array)

# __Exercise c__
tau = 10.3
time = np.linspace(0, seconds, seconds+1)
plt.plot(time, atoms_over_time)
plt.plot(time, number_of_atoms*np.exp(-time/tau))
plt.axis([0, seconds, 0, number_of_atoms])
plt.savefig("fig_random_decay.pdf")
```



Oppgave 8.3

Siden vi generer ganske mange verdier, er sannsynligheten veldig liten for at listen med treffvinkler blir tom. Har de sett bort i fra å sjekke om listen er tom, men fått til alt annet i oppgaven, er det helt greit å se bort i fra dette.

Det er heller ikke fullt så viktig om de husker å ta med en kommentar på om de ser forskjell i antall treffvinkler for a) og b).

```
import numpy as np
import random

def height(t,v0,alpha):
    g = 9.81
    return -.5*g*t**2 + v0*t*np.sin(alpha)

def find_hit_alphas(v0,b,h0,h1,alphas):
    hit_alpha = []
    for a in alphas:
        T = b/(v0*np.cos(a))
        y = height(T,v0,a)
        if (h0 <= y <= h1):
            hit_alpha.append(a)
    return hit_alpha

h0 = 3
h1 = 3.25
b = 3.5
g = 9.81

v0 = 25.
N = 1000
alphas = np.random.uniform(0,np.pi,N)

found_alphas_uni = find_hit_alphas(v0,b,h0,h1,alphas)
while(found_alphas_uni == []):
    alphas = np.random.uniform(0,np.pi,N)
    found_alphas_uni = find_hit_alphas(v0,b,h0,h1,alphas)


mean_alpha = np.mean(found_alphas_uni)
alphas = np.random.normal(mean_alpha,.05,N)
found_alphas_norm = find_hit_alphas(v0,b,h0,h1,alphas)

print "Number of found alphas using uniform: %d"%len(found_alphas_uni)
print "Number of found alphas using normal with mean = %g: %d"\
%(mean_alpha,len(found_alphas_norm))
```

```
>>> Number of found alphas using uniform: 20
>>> Number of found alphas using normal with mean = 0.755561: 329
```

Oppgave 8.4

Hele fremgangsmåten er ganske nøyaktig beskrevet i oppgaven. Resultatet av å plukke et random element i listen vår skal være en liste på 3 elementer.

I oppgave b) er det ikke krav om å implementere en testfunksjon. Det skal bare summes over listen, og sammenlignes med analytisk verdi.

```
import numpy as np
import random
import matplotlib.pyplot as plt

# __Exercise a__
m = 1e-22
T = 300.0
k = 1.38e-23
particles = 20
shape = (particles, 3)

s = np.sqrt(k*T/m)
vel = np.random.normal(0, s, shape)
print random.choice(vel)

# __Exercise b__
E_k = 0
for i in range(particles):
    E_k += 0.5*m*( vel[i,0]**2 + vel[i,1]**2 + vel[i,2]**2 )
print E_k, 3/2.0*k*T*particles
```

```
>>> [ -0.35447337  -4.79488557 -11.6815135 ]
>>> 1.21330434569e-19 1.242e-19
```

Oppgave 8.5

Her skal de bli kjent med fordelene ved å bruke vektorisering. Det nye er tidstakingen.

Ellers, er det ikke noe annet som er spesielt ved denne oppgaven. Hvis de har fått til enten a eller b, men ikke begge, så mister oppgaven litt hensikt - meningen er tross alt å bli kjent med forskjellene mellom vektoriserte funksjoner og 'standarde' funksjoner.

```
from random import uniform
from math import sqrt
import numpy as np
import time

C = 0.47
rho_w = 1000     # kg/m^3
rho = 1.293        # kg/m^3
g = 9.81

N = 100000

# a)
avg_vT = 0

time_start = time.time()
for i in range(N):
    r = uniform(0.001,0.006)
    vT_tmp = sqrt((8.*rho_w*r*g)/(3*rho*C))
    avg_vT += vT_tmp/N

time_used = time.time() - time_start
print "Average vT = %g m/s"%avg_vT
print "Time used w/o numpy: %g seconds"%time_used

# b)
time_start = time.time()
r = np.random.uniform(0.001,0.006,N)
avg_vT = np.sum(np.sqrt((8.*rho_w*r*g)/(3*rho*C)))/N
time_used = time.time() - time_start
print "Average vT = %g m/s"%avg_vT
print "Time used with numpy: %g seconds"%time_used
```

Resultat etter en kjøring (kun veiledene - verdiene kan jo variere):

```
>>> Average vT = 11.9873 m/s
    Time used w/o numpy: 0.237353 seconds
>>> Average vT = 11.9739 m/s
    Time used with numpy: 0.00913 seconds
```

Oppgave 9.1

Ideelt skal `ConstantAcceleration` brukes så mye som mulig. Dette betyr at den lagrer alle mulig verdier, og kalles på både i utregningen av posisjon og hastighet.

```
from constant_acceleration_class import ConstantAcceleration

class LinearAcceleration1(ConstantAcceleration):  # 'Is-a' relationship
    def __init__(self, x0, v0, a0, j):
        ConstantAcceleration.__init__(self, x0, v0, a0) # Let's superclass store values.
        self.j = j

    def __call__(self, t):  # Class call returns position.
        j = self.j
        return ConstantAcceleration.__call__(self, t) + 1.0/6*j**3

    def velocity(self, t):  # Method for returning velocity.
        j = self.j
        return ConstantAcceleration.velocity(self, t) + 0.5*j**2

class LinearAcceleration2:  # 'Has-a' relationship
    def __init__(self, x0, v0, a0, j):
        self.const_acc = ConstantAcceleration(x0, v0, a0)  # Creates instance of class
        self.j = j                                         # to store values.

    def __call__(self, t):
        j = self.j
        return self.const_acc(t) + 1.0/6*j**3

    def velocity(self, t):
        j = self.j
        return self.const_acc.velocity(t) + 0.5*j**2
```

Oppgave 9.2

Det er viktig at de kaller på superklassen sin funksjon for å regne ut massetettheten.

```
# a)
class Solid:
    def __init__(self,volume):
        self.volume = volume

    def mass_density(self,mass):
        return mass/self.volume

# b)
class Iron(Solid):
    def __init__(self,volume,mass):
        Solid.__init__(self,volume)
        self.mass = mass

    def density(self):
        return Solid.mass_density(self,self.mass)


# c)
def test_density():
    body = Iron(0.1,787)
    expected = 7870.
    cal = body.density()

    assert abs(cal - expected) < 1e-14, 'Got: %g, expected: %g'\
    %(cal,expected)

if __name__ == '__main__':
    test_density()
```

Oppgave 9.3

Poenget her er at de skal bruke `Cylinder` sin `inertia` for å beregne treghetsmomentet til sylinderskallet.

```
# Definition expected in a)
class GeometricShape:
    def __init__(self,mass):
        self.mass = mass

# Definition expected in b)
class Cylinder(GeometricShape):
    def __init__(self,mass,r):
        GeometricShape.__init__(self,mass)
        self.r = r

    def inertia(self):
        M = self.mass
        R = self.r
        return .5*M*R**2

# Definition expected in c)
class CylindricalShell(Cylinder):

    def inertia(self):
        return 2*Cylinder.inertia(self)

if __name__ == '__main__':
    # usage expected in a)
    shape = GeometricShape(5)

    # usage expected in b)
    cylinder = Cylinder(5,0.75)
    print "Cylinder has moment of inertia I = %g"%cylinder.inertia()

    # usage expected in c)
    cylinder_s = CylindricalShell(5,0.75)
    print "Cylindrical shell has moment of inertia I = %g"%cylinder_s.inertia()
```

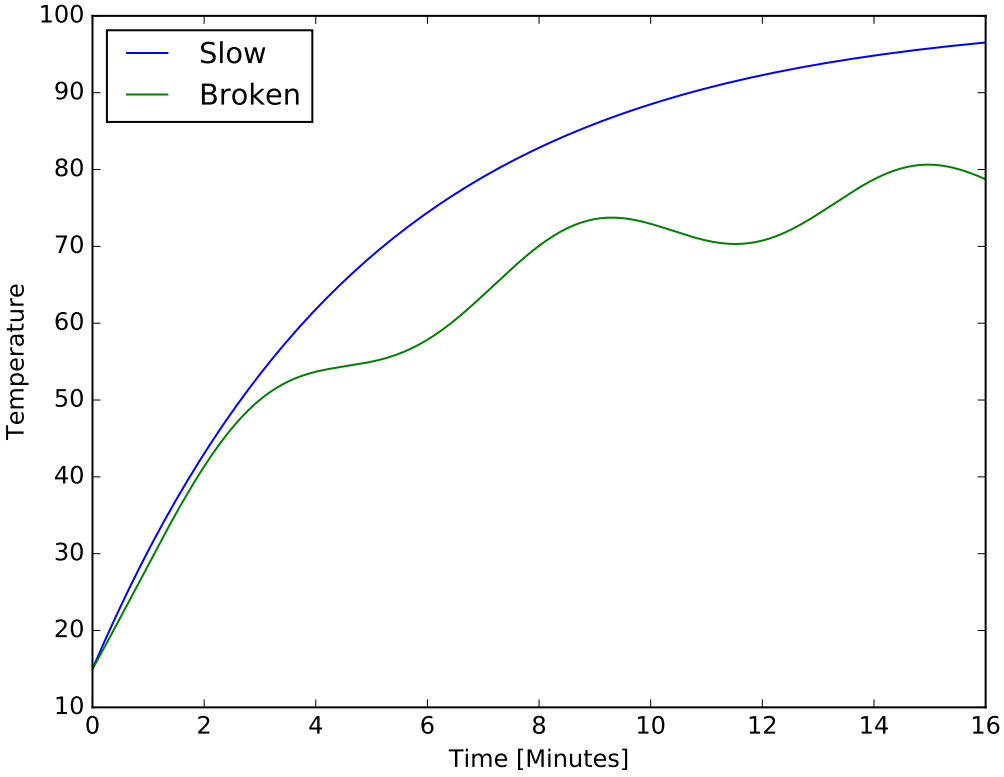
```
>>> Cylinder has moment of inertia I = 1.40625
>>> Cylindrical shell has moment of inertia I = 2.8125
```

Oppgave E.1

Utfordringen ligger nok i hvordan de skal sette opp systemet dersom temperaturen til vannkokeren endrer seg. Ellers, er det viktig at de får med `legend` for å kunne skille mellom når vannet er i den ødelagte vannkokeren eller ei.

```
import ODESolver
import numpy as np
import matplotlib.pyplot as plt

# a)
def temp_kettle(t):
    return 100

def f(u,t):
    k = 0.2
    t_env = temp_kettle(t)
    return -k*(u-t_env)

# b)
def temp_broken_kettle(t):
    return 20*np.sin(np.pi/3*t)+80

def f_broken(u,t):
    k = 0.2
    t_env = temp_broken_kettle(t)
    return -k*(u-t_env)


U0 = 15
T = 15      # Minutes
N = 10000

t_points = np.linspace(0,T+1,N)

solver = ODESolver.RungeKutta4(f)
solver.set_initial_condition(U0)
u,t = solver.solve(t_points)

temp_water = u

solver = ODESolver.RungeKutta4(f_broken)
solver.set_initial_condition(U0)
u,t = solver.solve(t_points)

temp_water_broken = u

plt.plot(t,temp_water,t,temp_water_broken)
plt.legend(['Slow','Broken'],loc='upper left')
plt.xlabel('Time [Minutes]')
plt.ylabel('Temperature')
plt.show()
```

(Plot er på neste side)


Slow
Broken
Temperature
Time [Minutes]
100
90
80
70
60
50
40
30
20
10
0
2
4
6
8
10
12
14
16

Oppgave E.2

ODE'en i a) er ganske straight forward. Bare å definere en funksjon for $Q(t)$, sette initialverdi, og løse.

I b) må Q omdefineres til å inneholde en if-test, som bestemmer V for forskjellige tidspunkt. Utover det gjøres alt på indentisk måte.

```
from numpy import sin, exp, linspace
import matplotlib.pyplot as plt
from ODESolver import ODESolver, ForwardEuler, RungeKutta4

Q0 = 0  # Initial electric charge, in Colomb
R = 1e8  # Resistance in Ohm
C = 2e-8  # Capasitance in Farad
timepoints = 101
seconds = 10
time_array = linspace(0, seconds, timepoints)
V0 = 8  # Voltage of battery when on

def Q_der(Q,t):  # Definition of Q'(t)
    return (V0 - Q/C)/R

# __Exercise a__
def Q_exact(t):  # Q(t) exact (only for charge-up).
    return (Q0 - C*V0)*exp(-t/(R*C)) + C*V0

circuit_FE = ForwardEuler(Q_der)
circuit_FE.set_initial_condition(Q0)
Q_FE, t = circuit_FE.solve(time_array)

circuit_RK4 = RungeKutta4(Q_der)
circuit_RK4.set_initial_condition(Q0)
Q_RK4, t = circuit_RK4.solve(time_array)

plt.plot(t, Q_FE, label='$Q(t)$, Forward Euler')
plt.plot(t, Q_RK4, label='$Q(t)$, Runge Kutta 4')
many_time_steps = linspace(0, seconds, 1001)  # For analytical solution.
plt.plot(many_time_steps, Q_exact(many_time_steps), label='$Q(t)$, exact')
plt.xlabel('Time [seconds]')
plt.ylabel('Electric charge [colomb]')
plt.title('Charging capasitor')
plt.legend()
plt.savefig('fig_E2.1.pdf')
plt.clf()

# __Exercise b__
def Q_der(Q,t):  # Redefining Q'(t) for exercise b
    if t < 10:
        V = 8
    elif t < 20:
        V = 4*sin(t) + 4
    else:
        V = 0
    return (V - Q/C)/R

time_array = linspace(0, 30, 101)
circuit_RK4_b = RungeKutta4(Q_der)
circuit_RK4_b.set_initial_condition(Q0)
Q_RK_b, t = circuit_RK4_b.solve(time_array)

plt.plot(t, Q_RK_b)
plt.xlabel('Time [seconds]')
plt.ylabel('Electric charge [colomb]')
plt.title('Charging, oscilating, and discharging capasitor')
plt.savefig('fig_E2.2.pdf')
plt.clf()
```

Oppgave E.3

Her er det litt å passe på. Siden vi jobber nå i to dimensjoner (bevegelse langs x- og y-aksen), så må det passes på at det gjøres riktig oppsett av ODE-ene. Sidene som oppgaven referer til i boken, er veldig nyttige og hinter veldig mye til hvordan oppsettet skal være.

```
import numpy as np
import ODESolver
import matplotlib.pyplot as plt

class Problem:
    def __init__(self,U0,r,w):
        self.U0 = U0

        self.g = 9.81
        self.r = r                # given in m
        self.rho = 1.293          # kg/m^3
        self.C = 0.47
        self.m = 0.057            # given in kg
        self.w = w                # given in m/s

    def __call__(self,u,t):
        x,vx,y,vy = u
        return [vx,-(self.rho*self.r**2*np.pi*self.C*(vx-self.w)**2)/(2*self.m),vy,-self.g
            ]

if __name__ == '__main__':

    x0 = 0; y0 = 0

    alpha = np.pi/4
    v0 = 10.

    U0 = [x0,v0*np.cos(alpha),y0,v0*np.sin(alpha)]
    T = 0.5
    N = 1000

    t_points = np.linspace(0,T+1,N)

    labels = []
    plt.hold('on')
    for w in [-10,-5,0,5,10]:
        problem = Problem(U0,0.03275,w)

        solver = ODESolver.RungeKutta4(problem)
        solver.set_initial_condition(problem.U0)

        u,t = solver.solve(t_points)

        x = u[:,0]
        y = u[:,2]
        labels.append('w = %g'%w)
        plt.plot(x,y)
    plt.xlabel('Position along ground [m]')
    plt.ylabel('Height [m]')
    plt.legend(labels)
    plt.savefig('fig_air_resistance.pdf')
    plt.show()
```

Oppgave E.4

En vanlig feil vil sannsynligvis være å konvertere noen av enhetene tilbake til sekunder/meter/kg, som da gir gale resultater, med mindre man bytter alt tilbake til SI enheter. Dette vil gi rimelig gale resultater, men viser vel mest mangel på forståelse for fysikk, og ikke programmering.

Å implementere 2. ordens ODEs kan være en del mindre intuitivt å sette opp en 1. ordens, men det er egentlig bare å følge oppskriften gitt i boka.

```
from numpy import sqrt, pi, linspace
import matplotlib.pyplot as plt
from ODESolver import ODESolver, ForwardEuler, RungeKutta4

def f(u,t):
    M = 1  # SolarMasses
    G = 4*pi**2  # AU^3/(yr^2*SM)
    x, y, vx, vy = u
    dx = vx
    dy = vy
    radius = sqrt(x**2 + y**2)
    dvx = -G*M*x/radius**2
    dvy = -G*M*y/radius**2
    return [dx, dy, dvx, dvy]

planet = RungeKutta4(f)
x = 1; y = 0  # AU
vx = 0; vy = 2*pi  # AU/yr
U0 = [x, y, vx, vy]
planet.set_initial_condition(U0)

time_points = linspace(0,10,1001)
u, t = planet.solve(time_points)

x, y, vx, vy = u[:,0], u[:,1], u[:,2], u[:,3]

plt.plot(x, y)
plt.axis('equal')
plt.show()
```