1 Løsningsforslag

Dette dokumentet inneholder løsningsforslag til oppgavene i kompendiet. Det anbefales selvfølgelig at du gjør et skikkelig forsøk på en oppgave før du ser på løsningsforslaget. Mer også at dette kun er løsnings*forslag*. Når man programmerer er det sjeldent bare en mulig vei frem til et gitt mål, og det er ofte man kan finne ganske forskjellige fremgangsmåter.

Merk også at disse løsningsforslagene ikke nødvendigvis er feilfrie, det kan ha sneket seg inn skrivefeil eller andre slurverier. Om du er i tvil om noe, eller mener noe ikke fungerer slik det står skrevet her er, så er det selvfølgelig bare å ta kontakt.

1.1 Variabler og Utregninger

Løsningsforslag til Oppgave 1 (Printing)

a)

```
print("Hei, Verden!")
```

```
Hei, Verden!
```

b) Her må vi først definere en tekstvariabel med navnet vårt, og deretter kan vi bruke print-formattering for å "flette" variabelen inn i en besked:

```
navn = "Ola"
print(f"Hei på deg {navn}!")
```

```
Hei på deg Ola!
```

c) Vi bruker `input()` til å spørre brukeren om et navn, det er også viktig å huske å lagre svaret i en variabel, ellers kan vi ikke bruke det senere

```
navn = input("Hva heter du? ")
print(f"Hei {navn}, hyggelig å møte deg!")
```

Løsningsforslag til Oppgave 2 (Volumberegninger)

a) For å få tilgang til π kan vi enten definere den selv, eller importere den. I første omgang bryr vi oss ikke så mye om enheter, og regner derfor i meter og m^3. For å skrive ut bruker vi vitenskapelig notasjon ved å spesifisere stilen med en `e`.

```
from math import pi
R = 6371000 # m
V = 4*pi*R**3/3
print(f"Jorda har et volum på {V:.2e} m^3.")
```

```
Jorda har et volum på 1.08e+21 m^3.
```

b) Løsningen blir veldig lik, vi bare endrer enheten på radiusen til dm, slik at volumet blir dm^3, som tilsvarer liter.

```
Jorda har et volum på 1.08e+24 liter.
```

c) For å finne den gjennomsnittelige massetettheten må vi definere massen som en variabel, og regne ut M/V:

```
M = 5.972e24
print(f"Gjennomsnittelig massetetthet: {M/V:.1f}
   kg per liter.")
```

```
Gjennomsnittelig massetetthet: 5.5 kg/liter.
```

Dette høres ut som en veldig rimelig gjennomsnittlig massetetthet. Om vi går ett steg lenger og Googler "fasitsvaret" finne vi 5.51 g/cm^3.

Løsningsforslag til Oppgave 3 (Regne ut pH)

a) Merk at variabelnavn i Python bare kan inneholder bokstaver, understrek (_) og tall, så vi kan ikke kalle konsentrasjonen $[H^+]$ i koden vår, vi velger derfor isteden å kalle den `konsentrasjon`. Vi velger også å skrive ut med stilen `:.2g`, bokstaven gbetyr at konsentrasjonen skrives ut som et desimaltall om det er passende, avhengig av størrelsesorden.

```python
pH = 3.6
konsentrasjon = 10**-pH
print(f"pH = {pH:.1f} ---> [H+] = {konsentrasjon:.2g} mol/L")
```

```
pH = 3.6 --->  [H+] = 0.00025 mol/L
```

b) For å regne ut konsentrasjonen trenger vi briggisk logaritme, altså logaritme av base 10. I matematikk skrives denne ofte bare som log, men i programmering er log den naturlige logaritmen. Da bruker vi istedet funksjonen `log10`, for å tydeliggjøre basen.

```python
from pylab import log10

konsentrasjon = 4.7e-5 # mol/L
pH = -log10(konsentrasjon)
print(f"[H+] = {konsentrasjon:.2g} mol/L ---> pH = {pH:.1f}")
```

```
[H+] = 4.7e-05 mol/L ---> pH = 4.3
```

c) Vi kan nå enten kjøre programmet vårt fra første deloppgave to ganger:

```
pH = 2.5 --->  [H+] = 0.0032
pH = 7.0 --->  [H+] = 1e-07
```

Og så dele det ene svaret på det andre. Eller vi kan skrive et nytt lite program som gjør hele beregningen:

```python
pH_cola = 2.5
pH_vann = 7.0

konsentrasjon_cola = 10**-pH_cola
konsentrasjon_vann = 10**-pH_vann

forhold = konsentrasjon_cola/konsentrasjon_vann

print(f"Cola har omtrent {forhold:.0f} ganger fler hydrogenioner enn vann.")
```

```
Cola har omtrent 31623 ganger fler hydrogenioner
   enn vann.
```

Løsningsforslag til Oppgave 4 (Vei-Fart-Tid)
Denne oppgaven handler om å sette opp en formel på papir først, og så kunne oversette den til kode. Samtidig må man holde styr på enheter.

a)

```python
v = 70 # km/h
t = 20 # min

s = v*(t/60)
print(f"Vi har kommet ca {s:.1f} km etter å ha
   reist i {t} minutter i {v} km/h.")
```

```
Vi har kommet ca 23.3 km etter å ha reist i 20
   minutter i 70 km/h.
```

Løsningsforslag til Oppgave 5 (Konsentrasjonskalkulator)

a) Programmet kan se ut som følger. Det viktigste i denne oppgaven er å holde styr på alle enhetene, da er kommentarer et godt virkemiddel.

```python
stoff = "NaCl"
M = ... # Molar masse av stoffet i g/Mol
V = ... # Volum i mL
c = ... # Ønsket konsentrasjon i mol/L

m = M*V*c # Masse av stoffet i mg
m /= 1000 # Regner om fra mg til g

# Skriv ut løsningen
print(f"{m:.3f} gram {stoff}")
print(f"Løst i {V} mL vann")
print(f"Gir en {c} mol/L løsning")
```

b) Når vi setter inn verdiene oppgitt i oppgaven får vi følgende output

```
14.61 gram NaCl
Løst i 100 mL vann
Gir en 2.5 mol/L løsning
```

Dette kan man sjekke høres rimelig ut ved å dobbeltsjekke beregningen for hånd, eller for eksempel å bruke en online konsentrasjonskalkulator.

c) Utifra svaret fra forrige oppgave ser vi at 14.6 gram gir en 2.5 mol/L løsning. Vi trenger altså litt mer en dobbelt så masse for å treffe metningspunktet. Prøver vi oss frem litt finner vi 6.1 mol/L som et godt estimat

```
35.648 gram NaCl
Løst i 100 mL vann
Gir en 6.1 mol/L løsning
```

Denne oppgaven er ment for å vise hvordan et program som allerede er skrevet kan brukes for å utforske andre problemstillinger en de man originalt var ute etter å løse. Dette spørsmålet kan også enkelt besvares med penn og papir, eller ved å skrive ny kode, men det hadde nok tatt lengre tid.

Løsningsforslag til Oppgave 6 (Hardy-Weinbergs Lov)
Selv om oppgaveteksten i dette tilfellet er veldig lang er det ikke så mye vi skal frem til i koden vår.

a) Hovedutfordringen er å få til $p + q = 1$ testen

```python
p = 0.9
q = 0.1

if p + q == 1:
    print(f"Allelefrekvens av allele A: {p:.0%}")
    print(f"Allelefrekvens av allele a: {q:.0%}")
    print()
    print(f"Frekvens av genotype AA: {p**2:.0%}")
    print(f"Frekvens av genotype Aa: {2*p*q:.0%}")
    print(f"Frekvens av genotype aa: {q**2:.0%}")
```

```
else:
    print("Merk at p+q må være lik 1.")
    print("Velg andre verdier og prøv på nytt.")
```

Merk at vi her går for den mest intuitive måten å sjekke betingelsen $p + q = 1$, men det finnes selvfølgelig andre måter å gjøre det på. Merk også at man egentlig skal være litt forsiktig med å sjekke at desimaltall er lik en bestemt verdi, fordi datamaskiner kan gjøre små, små feil i utregningene sine, og plutselig får vi en feilmelding fordi $p+q = 1.0000000001$, og dermed er testen falsk. Om man ønsker å gjøre dette skikkelig kan vi bruke funksjonen `isclose`, som kommer i pylab:

```
from pylab import *

p = 0.3
q = 0.7

if isclose(p+q, 1):
    ...
```

Men dette er mer avansert enn nødvendig for VGS-nivå tenker vi.

b) For denne oppgaven kjører vi koden to ganger med ulike p- og q-verdier og finner:

```
Allelefrekvens av allele A: 90%
Allelefrekvens av allele a: 10%

Frekvens av genotype AA: 81%
Frekvens av genotype Aa: 18%
Frekvens av genotype aa: 1%
```

Når vi øker q til 0.3 må vi huske å redusere p tilsvarende. Om vi glemmer dette skal det derimot gå greit, fordi programmet vårt skal si ifra at $p + q \neq 1$. Vi får nå

```
Allelefrekvens av allele A: 70%
Allelefrekvens av allele a: 30%

Frekvens av genotype AA: 49%
Frekvens av genotype Aa: 42%
```

```
Frekvens av genotype aa: 9%
```

Så ved å sammenligne de to ser vi at selvom sannsynligheten for alvorlig sigdcelle øker, ser vi også at andelen bærere øker forholdsvis *mer*. Altså kan fordelen av malariaresistens utveie risikoen for alvorlig sigdcelle.

1.2 If-tester

Løsningsforslag til Oppgave 7 (abc-formelen)

a) Den følgende koden skriver ut hvor mange løsninger ligningen har. Merk at det i noen sjeldne tilfeller vil bli litt feil, ettersom at d egentlig skal være 0 og vi har én løsning, men fordi vi har små avrundingsfeil på datamaskin får vi kanskje $d = 0.000000001$ eller lignende. Denne detaljen håndterer vi ikke her og nå.

```
a = ...
b = ...
c = ...

d = b**2 - 4*a*c

print(f"Ligningen {a}x^2 + {b}x + {c}x = 0.")
if d > 0:
    print("Har to løsninger.")
elif d == 0:
    print("Har en løsning.")
else:
    print("Har ingen løsninger")
```

b) Nå vil vi også skrive ut løsningene i tilegg til å si hvor mange de er. Her vil problemet med avrundingsfeil løse seg litt, for om d er ørlittegranne vekk fra 0, så vil de to røttene bli tilnærmet identiske.

```python
from pylab import sqrt

a = 1
b = 2
c = 1

d = b**2 - 4*a*c

print(f"Ligningen {a}x² + {b}x + {c}x = 0.")
if d > 0:
    x1 = (-b + sqrt(d))/(2*a)
    x2 = (-b - sqrt(d))/(2*a)
    print("Har to løsninger:")
    print(f"x1 = {x1}")
    print(f"x2 = {x2}")
elif d == 0:
    x0 = -b/(2*a)
    print("Har en løsning:")
    print(f"x0 = {x0}")
else:
    print("Har ingen løsninger")
```

```
Ligningen 1x² + 2x + 1x = 0.
Har en løsning:
x0 = -1.0
```

```
Ligningen 2x² + 3x + -9x = 0.
Har to løsninger:
x1 = 1.5
x2 = -3.0
```

```
Ligningen -0.5x² + 0x + 2x = 0.
Har to løsninger:
x1 = -2.0
x2 = 2.0
```

d) Det viktigste å huske på her er å konverte inputten fra brukeren til desimaltall ved hjelp av `float()`

```
print(f"Vi skal løsge ligningen: ax² + bx + cx = 0.")
a = float(input("Hva er a? "))
b = float(input("Hva er b? "))
c = float(input("Hva er c? "))
```

Løsningsforslag til Oppgave 8 (Bohr's Atommodell)

a)

```
n = ...
m = ...

E0 = 13.6

E = E0*(1/m**2 - 1/n**2)

print(f"Sprang fra {n} -> {m}")
print(f"ΔE = {E:.1f} eV")
```

```
n = ...
m = ...

E0 = 13.6
hc = 1240

E = E0*(1/m**2 - 1/n**2)
λ = 1240/E

print(f"Sprang fra {n} -> {m}")
print(f"ΔE = {E:.1f} eV")
print(f"λ = {λ:.0f} nm ")
```

b)

```
if λ < 380:
    farge = "Ultraviolett"
elif λ < 450:
    farge = "Fiolett"
elif λ < 485:
    farge = "Blått"
elif λ < 500:
    farge = "Turkis"
elif λ < 560:
    farge = "Grønnt"
elif λ < 590:
    farge = "Gult"
elif λ < 625:
    farge = "Oransjt"
elif λ < 740:
    farge = "Rødt"
else:
    farge = "Infrarødt"

print(f"Farge: {farge}")
```

d) For å finne de ulike bølgelengdene i Balmerserien kan vi nå kjøre programmet en rekke ganger, og endre n hver gang. Eller, om vi vil løse det med kode så kan vi legge all koden vår så langt i en funksjon `Bohr(n, m)` og så bruke en løkke til å kalle på den flere ganger:

```
def Bohr(n, m):
    ...

for n in range(3, 7):
    Bohr(n, 2)
    print()
```

```
Sprang fra 3 -> 2
ΔE = 1.9 eV
λ = 656 nm
Farge: Rødt

Sprang fra 4 -> 2
```

```
ΔE = 2.5 eV
λ = 486 nm
Farge: Turkis

Sprang fra 5 -> 2
ΔE = 2.9 eV
λ = 434 nm
Farge: Fiolett

Sprang fra 6 -> 2
ΔE = 3.0 eV
λ = 410 nm
Farge: Fiolett
```

Dette er de 4 linjene som er > 400 nm, som regnes som de synligelinjene i Balmer serien, derimot gir $7- > 2$ og $8- > 2$ også linjer som er synlig for de fleste med nakent øye.

e) Vi skal nå sammenligne med bildet som er vist. Dette hadde strengt tatt vært lettere om bilder hadde en akse som viste hvilke bølgelengder det var snakk om. Men vi ser en tydelig rød linje ($3 \rightarrow 2$), en tydelig blå/turkis ($4 \rightarrow 2$), og to tydelige fiolette ($5 \rightarrow 2$ og $6 \rightarrow 2$) Vi ser også en siste svak linje ($7 \rightarrow 2$).

1.3 Løkker

Løsningsforslag til Oppgave 9 (Trekanttall)

a)

```
T = 0
for i in range(1, 1001):
    T += i
print(T)
```

```
500500
```

```
n = 1000
T = n*(n+1)/2
print(T)
```

```
500500
```

Løsningsforslag til Oppgave 10 (Riskorn og Sjakkbrett)

a) Vi velger nå å skrive løkken som `for n in range(64)`. Da blir $n = 0, 1, \ldots 63$. Altså begynner vi å telle på 0. Dette er informatiker-måten å telle på. Da vil antall riskorn i rtuen være gitt ved 2^n, fordi første rute får $2^0 = 1$ riskorn som forventet.

```
total = 0
for n in range(64):
    rute = 2**n
    total += rute
print(total)
```

```
18446744073709551615
```

Vi kunne selvfølgelig løkket fra og med 1, da hadde vi skrevet `for n in range(1, 65)`, og latt antall riskorn i ruta vært gitt ved 2^{n-1}.

b)

```
print(f"{total:.2e}")
```

```
1.84e+19
```

Et veldig stort tall!

c) Vi endrer nå til en while-løkke, der betingelsen er `while total < 500 00`. Det vi må huske på da er å "manuelt" oppdatere n på innsiden av løkka.

```
total = 0
n = 0

while total < 50000:
    total += 2**n
    n += 1

print(f"Du trenger {n} felt")
print(f"Da har du {total} riskorn")
print(f"eller omtrent {total/50e3} kg ris")
```

```
Du trenger 16 felt
Da har du 65535 riskorn
eller omtrent 1.3107 kg ris
```

Løsningsforslag til Oppgave 11 (Fibonaccirekka)
Oppgaven var kanskje ikke formulert veldig tydelig. Poenger er ihvertfall å skrive ut Fibonacci-rekka, dette kan gjøres på flere måter, men den følgende koden får det ihvertfall til.

```
a = 0
b = 1

while b < 100:
    print(b)
    ny = a + b
    a = b
    b = ny
```

```
1
1
2
3
```

```
5
8
13
21
34
55
89
```

Om vi skriver ut tall under 1000 blir de siste tre tallene

```
377
610
987
```

Da estimerer vi det gyldne snitt til

```
print(987/610)
```

```
1.618032786885246
```

Om vi slår opp ϕ ser vi at dette stemmer til og med den 5 desimalen. Om vi øker hvor mange ledd i rekka vi finner vil vi få en enda bedre tilnærming.

Løsningsforslag til Oppgave 12 (Skuddår)
Vi slår sammen de første deloppgavene og viser et komplett program. Man kan enten jobbe med if-tester inne i andre if-tester. Isåfall må vi passe særlig på innrykkene:

```
år = 2016

if år % 4 == 0:
    if år % == 100:
        if år % 400:
            print(f"{år} er et skuddår.")
        else:
            print(f"{år} er ikke et skuddår")
    else:
else:
    print(f"{år} er ikke et skuddår
```

Eller vi kan bruke elif-blokker. Isåfall må vi passe på rekkefølge, fordi om én test slår-inn, gjør ikke de som følger etter det, altså må vi sjekke 400 først, så 100, så 4 til slutt:

```
if år % 400 == 0:
    print(f"{år} er et skuddår.")
elif år % 100 == 0:
    print(f"{år} er ikke et skuddår.")
elif år % 4 == 0:
    print(f"{år} er et skuddår.")
else:
    print(f"{år} er ikke et skuddår.")
```

Her er nok den siste måten å gjøre ting på både lettere å skrive, og lettere å forstå. Derimot må man som sagt skjønne at rekkefølgen er viktig. Merk også at man her trenger `elif` ("else-if"), ellers blir det feil.

For å regne ut gjennomsnitssdager per år er det viktig å inkludere riktig antall døgn. Ettersom at det finnes en regel for 400 år må vi altså velge et antall år delelig med 400. Vi velger intervallet $[0, 2000)$, som består av $2000 = 5 \cdot 400$ år. Vi kunne også valgt årene $[1, 2000]$.

For å regne gjennomsnittlig antall dager, finner vi først antall dager totalt med en løkke, og så regner vi ut antall dager delt på antall år.

```
antall_år = 2000

dager = 0
for år in range(antall_år):
    if år % 400 == 0:
        dager += 366
    elif år % 100 == 0:
        dager += 365
    elif år % 4 == 0:
        dager += 366
    else:
        dager += 365

print(f"Fra år 0 til {antall_år}")
print(f"Har det vært totalt {dager} dager")
print(f"Det utgjør {dager/antall_år} dager per år.")
```

```
Fra år 0 til 2000
Har det vært totalt 730485 dager
Det utgjør 365.2425 dager per år.
```

Vi finner at reglene slik de er formulert gir i snitt 365.2425, dette er altså så vidt litt høyere enn det tropiske året på 365.2422 døgn per år. Forskjellene er så små at man må begynne å snakke om andre detaljer som for eksempel at et døgn har blitt gradvis lengre over disse årene grunnet tidevannskrefter, som er grunnen til *skuddsekunder*, osv. Kort fortalt er altså regelene for skuddår ganske bra for det formålet de er laget.

Løsningsforslag til Oppgave 13 (Gangetabellen)
Her er hovedpoenget å forstå hvordan en løkke inne i en løkke fungerer.

Denne koden er forholdsvis kompakt og kompleks. Så for å forstå hvordan den virker kan vi gå ett steg av gangen. Når man jobber med løkker inne i hverandre er det ofte lurt å jobbe seg innenifra og ut. La oss derfor se på den innerste løkka først. For at den skal kjøre må vi sette `i` til en gitt verdi, la oss velge 4. I tilegg dropper vi den rare `end=""` biten av `print` i første omgang.

```
1 i = 4
2
3 for j in range(1, 11):
4     print(f"{i*j:5}")
```

Vi ser da at vi får skrevet ut 4-gangen

```
    4
    8
   12
  ...
   36
   40
```

Nå kan vi legge til `end=""`-biten igjen, og se hva som skjer. Da ser vi at vi får skrevet tallene på én linje, istedenfor under hverandre

```
   4    8   12   16   20   24   28   32   36   40
```

Det som skjer her er at alt vi skriver ut med `print()` vanligvis kommer på hver sin linje fordi funksjonen selv legger på et linjeskifte bak det vi skriver, ved å si `end=""` erstatter vi dette linjeskiftet med en tom streng, dvs, "ingenting", og følgelig havner ting på samme linje.

Når vi nå legger denne koden inn under en annen løkke der `i` varieres skriver vi altså først ut 1-gangen, så 2-gangen, så 3-gangen osv. Derimot får vi aldri noen linjeskift, så de havnet etter hverandre. Derfor legger vi til en tom `print()`, for å få et linjeskift mellom hver rad. Resultatet er en oversiktlig og ryddig tabell.

```
   1    2    3    4    5    6    7    8    9   10
   2    4    6    8   10   12   14   16   18   20
   3    6    9   12   15   18   21   24   27   30
   4    8   12   16   20   24   28   32   36   40
   5   10   15   20   25   30   35   40   45   50
   6   12   18   24   30   36   42   48   54   60
   7   14   21   28   35   42   49   56   63   70
   8   16   24   32   40   48   56   64   72   80
   9   18   27   36   45   54   63   72   81   90
  10   20   30   40   50   60   70   80   90  100
```

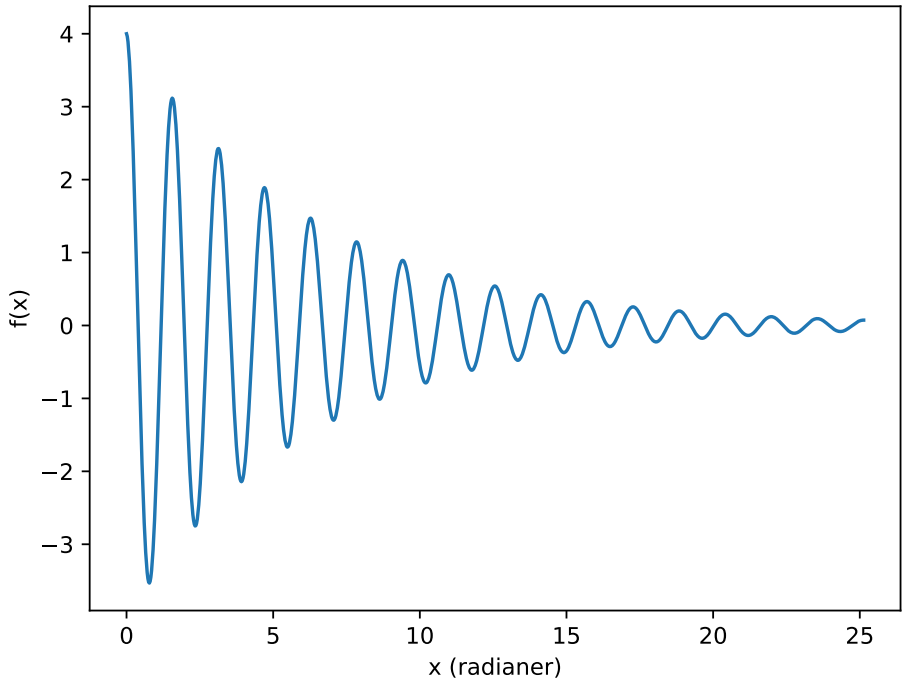
1.4 Plotting

Løsningsforslag til Oppgave 14 (Dempet svingning)
For å plotte denne funksjoner er det "enklest" å bruke *vektoriserte beregninger*, for da kan vi skrive ut utregningen så det ligner helt på det matematiske uttrykket. For å få til vektoriste beregninger kan vi benytte oss av `arange` eller `linspace` til å lage et *array*, en variabel som oppfører seg som en matematisk vektor. Følgende kode generer kurva og plotter den, i tilegg pynter vi litt på plotte med aksenavn, samt lagrer figuren som en .pdf-fil.

```
1 from pylab import *
2
3 x = linspace(0, 8*pi, 1001)
4 f = 4*cos(4*x)*exp(-x/(2*pi))
```

```
5
6  plot(x, f)
7  xlabel('x (radianer)')
8  ylabel('f(x)')
9  savefig('dempet_svingning.pdf')
10 show()
```

Resultatet blir som følger:



Løsningsforslag til Oppgave 15 (Parameterfremstilling av Sirkel)
Parameterfremstillingen for en sirkel med sentrum i origo er gitt ved

$$\begin{cases} x = r \cdot \cos(\theta), \\ y = r \cdot \sin(\theta), \end{cases}$$

der r er sirkelens radius, og vinkelen θ går fra 0 til 2π.

For å definere et annet sentrum må vi bare legge til en konstant til hver av koordinatene, vi velger å kalle disse x_0 og y_0:

$$\begin{cases} x = x_0 + r \cdot \cos(\theta), \\ y = y_0 + r \cdot \sin(\theta), \end{cases} .$$

Vi kan nå lage en funksjon som tegner en sirkel av radius r rundt (x_0, y_0) som følger:

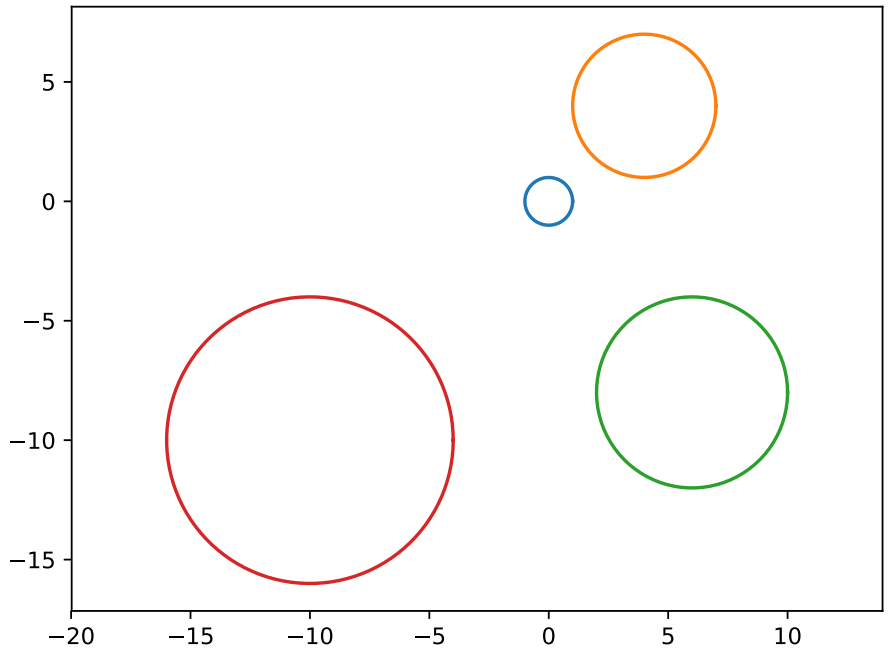
```
from pylab import *

def tegn_sirkel(r, x0, y0):
    theta = linspace(0, 2*pi, 1001)
    x = x0 + r*cos(theta)
    y = y0 + r*sin(theta)
    plot(x, y)
```

For å teste den velger vi plotte en del ulike sirkler i samme figur:

```
tegn_sirkel(1, 0, 0)
tegn_sirkel(3, 4, 4)
tegn_sirkel(4, 6, -8)
tegn_sirkel(6, -10, -10)

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



Selv om det på ingen måte er en del av oppgaven kan vi gå et steg lenger å tegne OL-ringene. Her bruker vi et par ting vi ikke har nevnt i kompendiet. For det første plotter vi nå med en bestemt farge, ved hjelp av `color='...'` argumentet til `plott`. I tilegg bruker vi `axis('off')` for å skru av alt av akser.

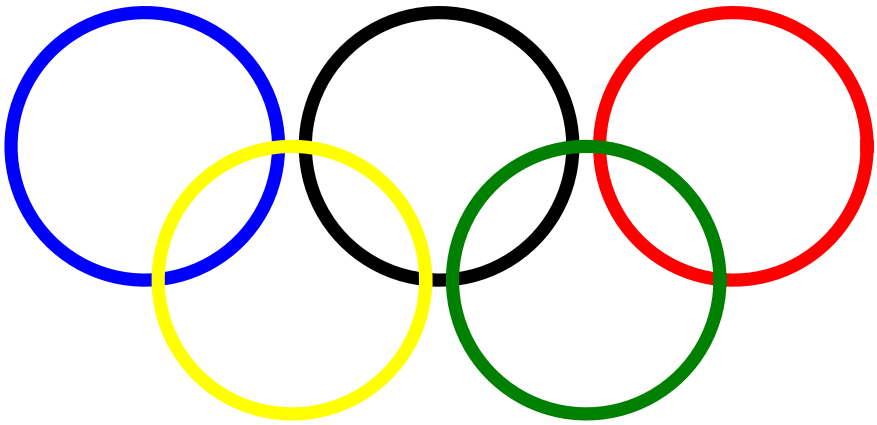
```
from pylab import *

def tegn_sirkel(r, x0, y0, farge):
    theta = linspace(0, 2*pi, 1001)
    x = x0 + r*cos(theta)
    y = y0 + r*sin(theta)
    plot(x, y, color=farge, linewidth=5)

tegn_sirkel(1,   0, 1, 'blue')
tegn_sirkel(1, 1.1, 0, 'yellow')
tegn_sirkel(1, 2.2, 1, 'black')
tegn_sirkel(1, 3.3, 0, 'green')
```

```
13 tegn_sirkel(1, 4.4, 1, 'red')
14
15 axis('equal')
16 axis('off')
17 savefig('OL_ringer.pdf')
18 show()
```

Da får vi følgende figur:



Løsningsforslag til Oppgave 16 (Grafisk løsning av ligninger)
Følgende kode tegner de to kurvene i samme figur:

```
from pylab import *

x = linspace(-5, 5, 1001)
f = x**2 - 2*x - 4
g = -2*x**2 - 4*x + 8

plot(x, f, label='f(x)')
plot(x, g, label='g(x)')
legend()
show()
```

Verdiene som er valgt gir ikke kryssningspunkter som er særlig elegante, ved å lese av finner vi omtrent $x = -2.4$ og $x = 1.7$. Du kan zoome inn på figuren om du ønkser å lese av mer nøyaktig enn dette.

For å dobbelsjekke svaret vårt kan vi skrive om ligningen:

$$f(x) = g(x) \quad \Rightarrow \quad f(x) - g(x) = 0.$$

Å finne kryssningspunktene er det samme som å finne røttene av ligningen $f(x) - g(x)$ som også er en annengradsligning:

$$f(x) - g(x) = 3x^2 + 2x - 12.$$

Om vi plugger dette inn i vårt abc-program fra en tidligere oppgave får vi:

```
Ligningen 3x^2 + 2x + -12x = 0.
Har to løsninger:
x1 = 1.694254176766073
x2 = -2.36092084343274
```

I dette tilfellet får vi stygge nullpunkter. Men om man ønsker er det lett å velge verdier som gir mer elegante nullpunkter, da må man bare velger koeffesienter så $f(x) - g(x)$ får fine nullpunkter.

1.5 Funksjoner

Løsningsforslag til Oppgave 17 (Absoluttverdi)

a)

```
def absoluttverdi(x):
    if x >= 0:
        return x
    else:
        return -x
```

Denne funksjonen vil gjøre jobben. La oss teste med et par tall. For å teste bør vi sjekke minst ett positivt tall og ett negativttall, samt 0, men gjerne flere verdier enn dette også:

```
for x in -5, -3, 0, 2, 4, 1e3:
    print(f"|{x}| = {absoluttverdi(x)}")
```

```
|-5| = 5
|-3| = 3
|0| = 0
|2| = 2
|4| = 4
|1000.0| = 1000.0
```

```
def størst_abs(a, b):
    if absoluttverdi(a) >= absoluttverdi(b):
        return a
    else:
        return b
```

Og igjen bør vi selvfølgelig teste:

```
print(størst_abs(2, 4))
print(størst_abs(0, 5))
print(størst_abs(0, -5))
print(størst_abs(-5, -3))
```

```
4
5
```

```
-5
-5
```

Merk at slik oppgaven er formulkert blir det riktig at de to siste tallene er -5, fordi oppgaven spør om selve tallet som har den største absoluttverdien, og ikke absoluttverdien i seg selv.

Løsningsforslag til Oppgave 18 (Konvertering av temperatur)

a)

```
def F2C(F):
    return 5/9*(F-32)
```

```
F = 350
C = F2C(F)
print(f"{F:.0f} grader Fahrenheit = {C:.0f} grader
    Celsius")
```

```
350.0 grader Fahrenheit = 176.7 grader Celsius
```

b) Den motsatte ligningen blir

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

Koden blir dermed

```
def C2F(C):
    return 9/5*C + 32
```

```
print(f"Frysepunkt: {C2F(0):.0f} grader F")
print(f"Kokepunkt: {C2F(100):.0f} grader F")
```

```
Frysepunkt: 32 grader F
Kokepunkt: 212 grader F
```

Løsningsforslag til Oppgave 19 (Kombinatorikk)

a)

```python
def fakultet(n):
    total = 1
    for i in range(1, n+1):
        total *= i
    return total
```

b)

```python
print(f"5! = {fakultet(5)}")
```

```
120
```

Det er altså 120 rekkefølger man kan velge seg.

c)

```python
def nCk(n, k):
    return fakultet(n)/(fakultet(k)*fakultet(n-k))
```

d)

```python
print(f"28C3 = {nCk(28, 3)}")
```

```
28C3 = 3276.0
```

Det er altså 3276 måter å plukke de 3 elevene.

e)

```python
def nPk(n, k):
    return fakultet(n)/fakultet(n-k)
```

f)

```python
print(f"28P3 = {nPk(28, 3)}")
```

```
28P3 = 19656
```

Det er altså 19656 måter å plukke ut de 3 elevene om rekkefølge også er viktig.

Vi kan kjapt sjekke at svarene våres er internt konsistente: Det er $3! = 6$ måter å ordne 3 elever på, som om det er 3276 måter å plukke ut 3 ulike elever, vil det være $3276 \cdot 6$ mulige permutasjoner av 3 elever. Dette produktet blir 19656 som forventet.