Repetisjon/oppgaver i BED-1304 Python-Lab

Markus J. Aase

Forelesning 8 - SymPy

Informasjon

Til nå har vi gått igjennom flere ting i Python-lab. Det inkluderer temaer som:

- Python basics
- Funksjoner (Innebygde funksjoner og de vi definerer selv)
- Lister, Dictionary og Tuples
- Logikk og løkker (`if/else`-setninger, `for`-løkker og `while`-løkker).
- Pakker som `Pandas`, `NumPy` og `Matplotlib`

Videre skal vi fokusere på biblioteket `SymPy`, og bruken av det for å arbeide med symbolsk matematikk - hvor vi kan løse likninger, derivere og mye mer.

Dette dokumentet er lagd for å repetere og teste forståelsen av kjernepensum i **BED-1304 Python-Lab**. Husk at dette er et supplement til forelesning og seminar.

God koding!

Introduksjon til SymPy

Hva er SymPy?

`SymPy` er et bibliotek i Python for **symbolsk matematikk**. I motsetning til `NumPy`, som regner numerisk, kan SymPy manipulere matematiske uttrykk nøyaktig slik vi gjør på papir. Det betyr at vi kan løse ligninger, faktorisere uttrykk, derivere og integrere funksjoner, og mye mer.

Installasjon

Som alltid når vi bruker en ny pakke i Python, må den først installeres:

```
!pip install sympy
```

NB: I Jupyter Notebook kan det hende du må bruke % i stedet for ! foran kommandoen.

Bruk av pakken

Som alle pakker i Python, må vi *importere* pakken før vi kan bruke den. Det gjør vi på følgende vis,

```
import sympy as sp
```

og da kan vi bruke pakken. Vi starter med et enkelt eksempel.

Enkelt eksempel

La oss begynne med et lite eksempel hvor vi løser ligningen $x^2 = 9$:

```
import sympy as sp

# Definerer symbolet
x = sp.Symbol('x')

# Løser ligningen x^2 = 9
solution = sp.solve(sp.Eq(x**2, 9), x)
print(solution)
```

Output blir:

$$x = -3 \quad \text{eller} \quad x = 3$$

Dette viser hvordan SymPy kan brukes til å finne **eksakte løsninger** på ligninger.

Faktorisering

Vi kan faktorisere uttrykket $x^2 - 9$ direkte i SymPy:

```
faktor = sp.factor(x**2 - 9)
print(faktor)
```

Output:

$$x^2 - 9 = (x - 3)(x + 3)$$

Derivasjon

Derivasjon gjøres med `sp.diff()`:

```
f = x**2 - 9
df = sp.diff(f, x)
print(df)
```

Output:

$$\frac{d}{dx}(x^2 - 9) = 2x$$

Løsning av ligninger

Vi kan kombinere disse metodene for å finne nullpunkter. Det koden under her gjør er;

$$f(x) = 0$$

$$x^2 - 9 = 0$$

og deretter finner vi nullpunktene.

```
nullpunkter = sp.solve(sp.Eq(f, 0), x)
print(nullpunkter)
```

Output:

$$x = -3 \quad \text{eller} \quad x = 3$$

Eksempel på evaluering med `subs()`

Hvis vi ønsker å evaluere funksjonen i et punkt, kan vi bruke `subs()`:

```
# Her erstatter vi x med tallet 2
verdi = f.subs(x, 2)
print(verdi)
```

Output:

$$f(2) = 2^2 - 9 = -5$$

Integrasjon

I tillegg til derivasjon kan vi bruke SymPy til å finne **integraler**. Dette gjøres med funksjonen `sp.integrate()`.

```
import sympy as sp

x = sp.Symbol('x')

# Ubestemt integral
f = 3*x**2
F = sp.integrate(f, x)
print(F)
```

Output:

$$\int 3x^2 \, dx = x^3$$

Vi kan også beregne bestemte integraler ved å angi grensene for integrasjonen:

```
# Bestemt integral fra 0 til 2
bestemt = sp.integrate(x**2, (x, 0, 2))
print(bestemt)
```

Output:

$$\int_0^2 x^2 \, dx = \frac{8}{3}$$

Dermed kan vi bruke SymPy både til **ubestemte** og **bestemte integraler**, samt integraler over uendelige grenser.

Skriv matematiske uttrykk i Python

I SymPy kan vi skrive matematiske uttrykk nesten slik vi er vant til fra matematikk, men med noen få justeringer:

- Multiplikasjon må alltid skrives eksplisitt med `*`. Eksempel: $2x$ må skrives som `2*x`.
- Potenser skrives med `**`. Eksempel: x^2 skrives som `x**2`.
- Brøkstreker må skrives med divisjon `/`. Eksempel: $\frac{1}{x}$ skrives som `1/x`.
- Funksjoner som $\sin(x)$ og $\ln(x)$ finnes i `sympy`-biblioteket. Eksempel: `sp.sin(x)`, `sp.log(x)`.

```
import sympy as sp

x = sp.Symbol('x')

# Eksempler på uttrykk
f1 = 2*x + 3
f2 = x**2 - 4
f3 = sp.sin(x) + sp.log(x)
```

Kjører vi følgende celler, får vi

```
f1
```

$f_1 = 2x + 3$

```
f2
```

$f_2 = x^2 - 4$

```
f3
```

$f_3 = \sin(x) + \ln(x)$

Med disse enkle eksemplene har vi allerede sett hvordan SymPy kan brukes til **faktorisering, derivasjon, integrasjon, løsning av ligninger og evaluering av funksjoner**.

Likhet i SymPy

I Python betyr `=` alltid **tilordning av en verdi til en variabel**. Dette gjelder også når vi bruker `SymPy`. Hvis vi ønsker å sette opp en matematisk ligning, må vi derfor bruke andre verktøy.

`==` tester struktur, ikke matematikk

Tegnet `==` brukes i Python for å sjekke om to ting er like. I `SymPy` betyr dette en test av **strukturell likhet**, ikke matematisk likhet.

```
import sympy as sp

x = sp.symbols('x')
x + 1 == 4
# gir False
```

Her sjekker Python rett og slett: «Er objektet `x+1` nøyaktig det samme som tallet 4?». Svaret er selvsagt `False`.

Et litt mer spennende eksempel er:

```
(x + 1)**2 == x**2 + 2*x + 1
# gir False
```

Matematisk vet vi at uttrykkene er like, men strukturelt er de forskjellige (det ene er et kvadrat, det andre en sum av tre ledd).

Skrive ligninger: `sp.Eq`

For å skrive en ligning symbolsk i SymPy, må vi bruke funksjonen `sp.Eq()`:

```
sp.Eq(x + 1, 4)
# gir: x + 1 = 4
```

Dette lager et objekt som representerer en ligning, som vi kan bruke videre f.eks. med `sp.solve()`.

Teste matematisk likhet: `simplify`

Hvis vi vil sjekke om to uttrykk er **matematisk ekvivalente**, kan vi trekke det ene fra det andre og forenkle:

```
expr = (x + 1)**2 - (x**2 + 2*x + 1)
sp.simplify(expr)
# gir 0
```

Siden resultatet er 0, vet vi at uttrykkene er like.

Oppsummering

- `=` brukes til variabeltilordning i Python.
- `==` tester om to uttrykk er **strukturelt identiske**.
- `sp.Eq()` lager en **symbolsk ligning**.
- `sp.simplify()` kan brukes til å sjekke **matematisk likhet**.

Oversikt over funksjoner i SymPy

1. Definere symboler

Først må vi definere hvilke variabler vi ønsker å jobbe med. Dette gjøres med `sp.symbols()`.

```
import sympy as sp

x, y = sp.symbols('x y')
```

Nå kan x og y brukes i uttrykk og ligninger.

OBS: Vi kan bruke både `sp.Symbol()` og `sp.symbols()` for å definere symboler. Forskjellen er at `sp.Symbol()` brukes til å lage **ett symbol** om gangen, mens `sp.symbols()` kan brukes til å lage **ett eller flere symboler** samtidig.

```
# Definere ett symbol, en av gangen
a = sp.Symbol('a')
b = sp.Symbol('b')

# Flere symbol av gangen
x, y, z = sp.symbols('x y z')
print(x, y, z)
```

2. Ligninger med `sp.Eq`

For å sette opp en ligning i SymPy bruker vi `sp.Eq(venstre, høyre)`.

```
# Ligning: x^2 = 9
eq = sp.Eq(x**2, 9)
print(eq)
```

Dette gir:

$$x^2 = 9$$

3. Løse ligninger med `sp.solve`

Når vi har definert en ligning, kan vi løse den ved hjelp av `sp.solve()`.

```
solution = sp.solve(eq, x)
print(solution)
```

Løsningen blir:

$$x = -3 \quad \text{eller} \quad x = 3$$

4. Importere `solve` direkte

Man kan også importere `solve` direkte fra `sympy.solvers`, som kan være praktisk i større prosjekter.

```
from sympy.solvers import solve

solve(eq, x)
```

Løsningen her blir også:

$$x = -3 \quad \text{eller} \quad x = 3$$

5. Førsteordensbetingelser (derivasjon)

I økonomi brukes SymPy ofte til å finne maksimum og minimum av funksjoner. Dette gjøres ved å derivere og sette den deriverte lik null.

```
# Eksempel: Finn maksimum for f(x) = -x^2 + 4x
f = -x**2 + 4*x

# Deriverer funksjonen
f_prime = sp.diff(f, x)

# Setter f'(x) = 0
critical_points = sp.solve(sp.Eq(f_prime, 0), x)
print(critical_points)
```

Her finner vi det kritiske punktet:

$$f'(x) = -2x + 4 = 0 \quad \Rightarrow \quad x = 2$$

For å sjekke om dette er et maksimum eller minimum, kan vi se på den andrederiverte:

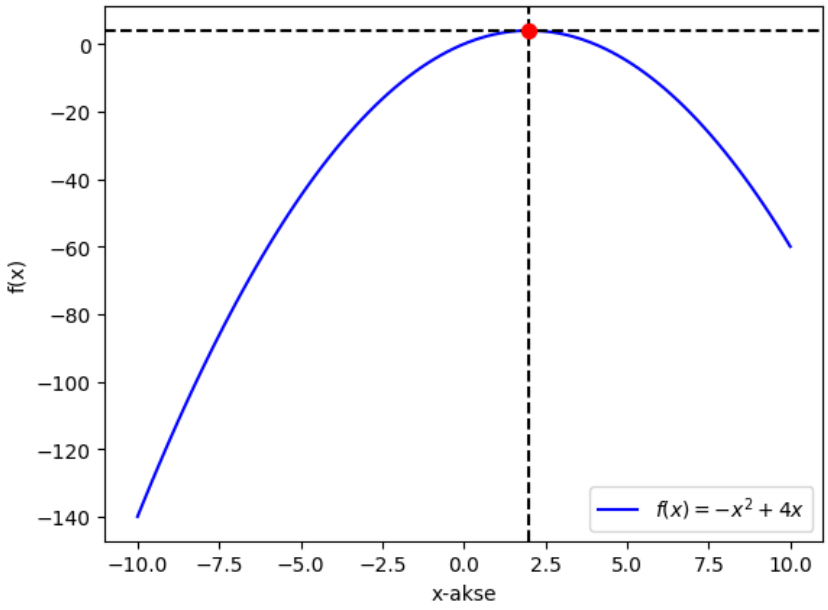
```
f_doubleprime = sp.diff(f_prime, x)
print(f_doubleprime.subs(x, 2))
```

Her får vi $f''(2) = -2 < 0$, altså et maksimum.

Nå kan vi visualisere funksjonen, for å se om dette stemmer. Da bruker vi det vi har lært om `matplotlib`.

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

x = np.linspace(-10, 10, 100)

def f_x(x):
    return -x**2 + 4*x

# Tegner funksjonen
plt.plot(x, f_x(x), color='blue', label="$f(x) = -x^2 + 4x$")

# Highlighter punktet (2,f(2))
plt.scatter(2, f_x(2), color='red', s=50, zorder=5)

# Aksetitler
plt.xlabel("x-akse")
plt.ylabel("f(x)")

# Horisontal linje i y=f(2)
plt.axhline(y=f_x(2), color='black', linestyle='--')

# Vertikal linje i x=2
plt.axvline(x=2, color='black', linestyle='--')

plt.legend()

plt.show()
```

Dette gir et output som dette:
Hvor det ser ut til å stemme at punktet $x = 2$ gir topp-punktet til funksjonen $f(x)$.



Figur 1: Plott av funksjonen $f(x) = -x^2 + 4x$

6. Bruk av `sp.subs()`

I SymPy kan vi evaluere et uttrykk ved å sette inn en bestemt verdi for en variabel. Dette gjøres med funksjonen `.subs()`. Det kommer fra engelsk *substitute*, for å substituere/erstatte.

```
import sympy as sp

# Definerer symbol
x = sp.Symbol('x')

# Definerer en funksjon
f = -x**2 + 4*x

# Setter inn x = 2
value = f.subs(x, 2)
print(value)
```

Output blir:

$$f(2) = -(2)^2 + 4 \cdot 2 = 4$$

Dette er spesielt nyttig når vi har funnet et kritisk punkt eller en løsning på en ligning, og ønsker å finne funksjonsverdien i dette punktet.

Eksempel: Bruk sammen med `solve()`

Vi kan kombinere `sp.solve()` og `.subs()` for å finne både løsning og funksjonsverdi:

```
# Finn kritisk punkt for f(x) = -x^2 + 4x
f_prime = sp.diff(f, x)
critical_point = sp.solve(sp.Eq(f_prime, 0), x)

# Setter inn i f(x) for å få maksverdien
max_value = f.subs(x, critical_point[0])

print("Kritisk punkt:", critical_point[0])
print("Maksimalverdi:", max_value)
```

Resultat:

$$x^* = 2, \quad f(x^*) = 4$$

Dermed finner vi at funksjonen har sitt maksimum ved $x = 2$ med en funksjonsverdi lik 4.

NB: Legg merke til at eksempelet over finner **ett kritisk punkt**, nemlig $(2,\ f(2)) = (2,\ 4)$. Men, hva om kodelinja

```
critical_point = sp.solve(sp.Eq(f_prime, 0), x)
```

gir tilbake en liste med mer enn ett punkt i lista. Eksempelet over, finner det første elementet i lista, det ser vi som følge av linja

```
max_value = f.subs(x, critical_point[0]) # Her henter vi første element, ved
    indeks 0!
```

Men hva som funksjonen er $f(x) = x^3 - 3x$, som vi vet har to kritiske punkt (topp- eller bunnpunkt), nemlig $x = 1$ og $x = -1$.

Det kan vi finne i SymPy ved å kjøre følgende kode som benytter seg av en `for`-løkke for å iterere gjennom alle de kritiske punktene i lista `critical_points`.

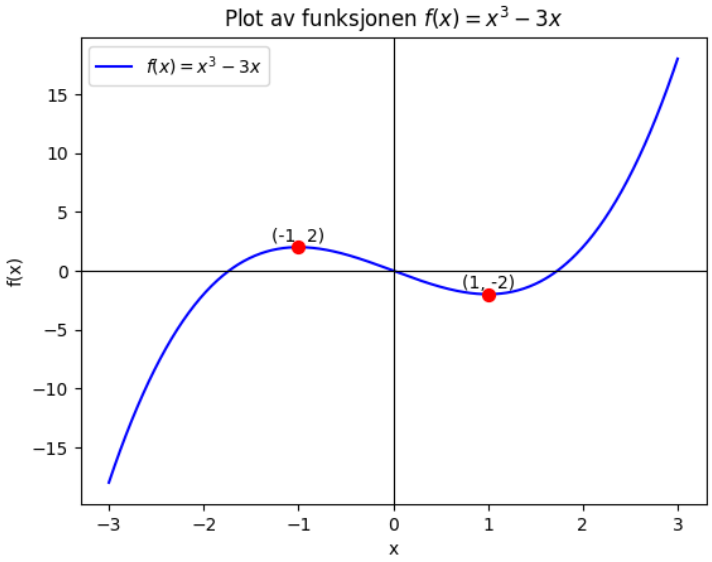
```
import sympy as sp

x = sp.Symbol('x')

# Eksempel: f(x) = x^3 - 3x
f = x**3 - 3*x

# Deriver funksjonen
f_prime = sp.diff(f, x)

# Finn alle kritiske punkt
critical_points = sp.solve(sp.Eq(f_prime, 0), x)

print("Kritiske punkt:", critical_points)

# Evaluer funksjonen i hvert kritisk punkt
for cp in critical_points:
    value = f.subs(x, cp)
    print(f"f({cp}) = {value}")
```

Som gir oss output:
Kritiske punkt: $[-1, 1]$
$f(-1) = 2$
$f(-1) = -2$

Dette kan vi plotte ved hjelp av `NumPy` og `matplotlib`.

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

x = np.linspace(-3, 3, 500)
y = x**3 - 3*x

plt.plot(x, y, label="$f(x) = x^3 - 3x$", color="blue")

# Markerer kritiske punkt
crit_points = [(-1, 2), (1, -2)]
for cp in crit_points:
    plt.scatter(cp[0], cp[1], color="red", s=50, zorder=5)
    plt.text(cp[0], cp[1]+0.5, f"({cp[0]}, {cp[1]})", ha="center")

plt.xlabel("x")
plt.ylabel("f(x)")
```

```
plt.title("Plot av funksjonen $f(x) = x^3 - 3x$")
plt.axhline(0, color="black", linewidth=0.8)
plt.axvline(0, color="black", linewidth=0.8)
plt.legend()
plt.show()
```



Figur 2: Plott av funksjonen $f(x) = x^3 - 3x$, med sine kritiske punkter.

7. Kjerneregelen med SymPy

Et nyttig triks i SymPy er å lage egne funksjoner som bruker de vanlige reglene fra kalkulus (matematikk). Et eksempel er **kjerneregelen** (chain rule), som sier at for en sammensatt funksjon

$$h(x) = f(g(x)),$$

har vi den deriverte

$$h'(x) = f'(g(x)) \cdot g'(x).$$

Dette kan vi kode selv i SymPy slik:

```
def f_deriv_chain(f, g):
    df = sp.diff(f)         # Deriverer f med hensyn til sitt argument
    dg = sp.diff(g)         # Deriverer g med hensyn til sitt argument

    return df.subs({'y': g}) * dg
```

Her antar vi at funksjonen f er skrevet som en funksjon av y, mens g er en funksjon av x. La oss ta et enkelt eksempel, f.eks. $f(g(x)) = (x^3)^2$:

```
x, y = sp.symbols('x y')

f = y**2          # f(y) = y^2
g = x**3          # g(x) = x^3

result = f_deriv_chain(f, g)
print(result)
```

Resultatet blir:

$$h'(x) = 2(x^3) \cdot 3x^2 = 6x^5,$$

som er den riktige deriverte av $h(x) = x^6$.

Oppsummering

- `sp.symbols()` brukes til å definere symboler (variabler) som vi kan bruke i uttrykk.
- `sp.Eq()` brukes til å definere ligninger.
- `sp.solve()` løser ligningene symbolsk.
- `solve()` kan også importeres direkte fra `sympy.solvers`.
- Førsteordensbetingelser (derivasjon og nullpunktsanalyse) brukes til å finne maksimum og minimum.
- `.subs()` brukes til å evaluere uttrykk ved å sette inn bestemte verdier.
- Vi kan implementere egne regler, som f.eks. **kjerneregelen**, direkte i SymPy.

Del 1: Flervalgsoppgaver - SymPy og økonomi

1. Hva gjør følgende kode?

```
import sympy as sp
x = sp.Symbol('x')
f = -x**2 + 4*x
f_prime = sp.diff(f, x)
print(f_prime)

```

   a) Printer $-x^2 + 4x$
   b) Printer $-2x + 4$
   c) Printer $f(2)$
   d) `Error`

2. Hvilken kommando brukes for å sette opp en ligning i SymPy?

   a) `sp.Eq()`
   b) `sp.solve()`
   c) `sp.subs()`
   d) `sp.Symbol()`

3. Hva gjør følgende kode?

```
x = sp.Symbol('x')
ettersporsel = 100 - 2*x
tilbud = 20 + 3*x
likevekt = sp.solve(sp.Eq(ettersporsel, tilbud), x)
print(likevekt)

```

   a) Løser tilbudsfunksjonen alene
   b) Finner likevektsmengden x^*
   c) Finner likevektsmengden og -prisen
   d) Feilmelding

4. Hva gjør denne koden?

```
f = x**2
print(f.subs(x, 3))

```

   a) Returnerer symbolet x
   b) Returnerer 9
   c) Returnerer 3
   d) Feilmelding

5. Når vi finner maksimum/minimum av en funksjon, hvilken prosedyre er korrekt?

   a) Sett den andrederiverte lik null
   b) Sett funksjonen lik null
   c) Sett den første deriverte lik null og sjekk fortegnet til den andrederiverte
   d) Sett inn tall tilfeldig

Del 2: Hva blir output? - SymPy og økonomi

1
```
import sympy as sp
x = sp.Symbol('x')
ettersporsel = 100 - 2*x
tilbud = 20 + 3*x
sp.solve(sp.Eq(ettersporsel, tilbud), x)
```

2
```
f = -x**2 + 4*x
kritisk = sp.solve(sp.Eq(sp.diff(f, x), 0), x)
f.subs(x, kritisk[0])
```

3
```
y = sp.Symbol('y')
f = y**2
g = sp.exp(x)
df_chain = sp.diff(f).subs({y: g}) * sp.diff(g)
print(df_chain)
```

Del 3 - Hva vi har lært til nå: SymPy-case

Her skal vi kombinere funksjoner, `SymPy`, og `matplotlib` i en økonomisk kontekst.

Oppgave

Vi ser på et enkelt tilbuds- og etterspørselsmarked:

$$Q_d = 100 - 2p, \qquad Q_s = 20 + 3p$$

hvor Q_d er etterspørsel, Q_s er tilbud og p er prisen.

1. Bruk `SymPy` til å finne likevektsprisen og -mengden.
2. Lag en funksjon som generaliserer: gitt en etterspørselsfunksjon og en tilbudsfunksjon, finn likevekten.
3. Bruk `Matplotlib` til å visualisere tilbuds- og etterspørselskurvene og marker likevektspunktet.
4. Ekstra utfordring: Anta at myndighetene innfører en skatt på 5 per enhet. Endre tilbudsfunksjonen og finn den nye likevekten. Visualiser endringen.

Løsningsforslag Del 1: Flervalgsoppgaver

1. b) Printer den deriverte $-2x + 4$
2. a) `sp.Eq()`
3. b) Finner likevektsmengden x^*
4. b) Returnerer 9
5. c) Sett den første deriverte lik null og sjekk fortegnet til den andrederiverte

Løsningsforslag Del 2: Hva blir output?

1. Løsning: $p = 16$. (Setter etterspørsel = tilbud).
2. Kritisk punkt: $x = 2$, funksjonsverdi: $f(2) = 4$.
3. (a) Først defineres en symbolsk variabel y.
   (b) Vi setter $f(y) = y^2$ som en funksjon av y.
   (c) Videre defineres $g(x) = e^x$.
   (d) Deretter beregnes den deriverte av f, som er

$$f'(y) = \frac{d}{dy} y^2 = 2y.$$

   (e) Vi setter inn $g(x)$ i stedet for y, altså

$$f'(g(x)) = 2e^x.$$

   (f) Til slutt multipliserer vi med den deriverte av $g(x)$:

$$g'(x) = \frac{d}{dx} e^x = e^x.$$

   (g) Hele uttrykket blir da

$$\frac{d}{dx}\big(f(g(x))\big) = f'(g(x)) \cdot g'(x) = (2e^x)(e^x) = 2e^{2x}.$$

Løsningsforslag Del 3: SymPy-case

Oppgave 1

```python
import sympy as sp
x = sp.Symbol('x')

Qd = 100 - 2*x
Qs = 20 + 3*x

likevekt_pris = sp.solve(sp.Eq(Qd, Qs), x)[0]
likevekt_mengde = Qd.subs(x, likevekt_pris)

print("Pris:", likevekt_pris)
print("Mengde:", likevekt_mengde)
```

Resultat:

$$p^* = 16, \quad Q^* = 68$$

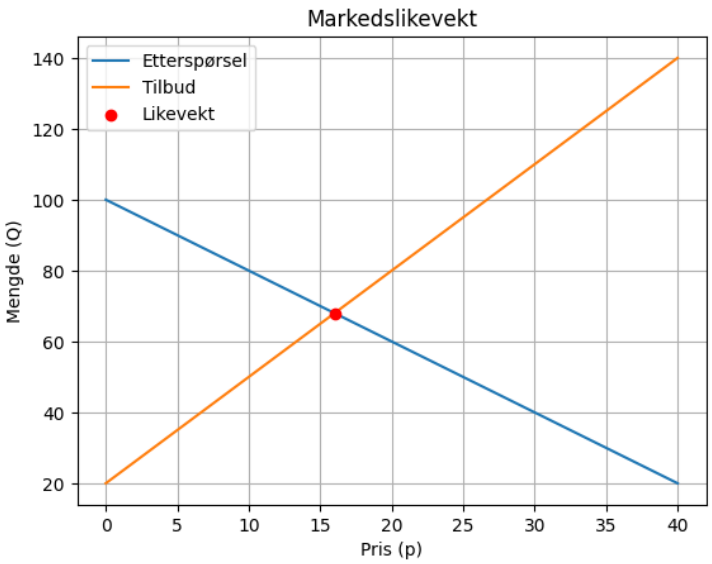
Oppgave 2

```python
def finn_likevekt(Qd, Qs, pris):
    likevekt_pris = sp.solve(sp.Eq(Qd, Qs), pris)[0]
    likevekt_mengde = Qd.subs(pris, likevekt_pris)
    return likevekt_pris, likevekt_mengde

pris = sp.Symbol('p')
Qd = 100 - 2*pris
Qs = 20 + 3*pris
finn_likevekt(Qd, Qs, pris)
```

Oppgave 3

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

# Plot funksjonene
p_values = np.linspace(0, 40, 100)
# Ekstraoppgave, finn ut nøyaktig hva de to linjene under her gjør.
Qd_values = [100 - 2*p for p in p_values]
Qs_values = [20 + 3*p for p in p_values]

plt.plot(p_values, Qd_values, label="Etterspørsel")
plt.plot(p_values, Qs_values, label="Tilbud")

# Marker likevekten
plt.scatter([16], [68], color="red", zorder=5, label="Likevekt")

plt.xlabel("Pris (p)")
plt.ylabel("Mengde (Q)")
plt.legend()
plt.title("Markedslikevekt")
plt.grid(True)
plt.show()
```

Hvor plottet ser noe sånn her ut:



Figur 3: Plott for del 3 (oppgave 3).

Oppgave 4 (ekstra)

Med skatt på 5 blir tilbudsfunksjonen:

$$Q_s = 20 + 3(p - 5)$$

```
Qs_skatt = 20 + 3*(pris - 5)
ny_pris, ny_Q = finn_likevekt(Qd, Qs_skatt, pris)

print("Ny pris:", ny_pris, "Ny mengde:", ny_Q)
```

For å visualisere endringene, kan du lage et plott som ligner på det fra forrige oppgave - men også plotter `Qs_skatt`. Det kan dere gjøre selv:)