Programmering i skolen:
Et kræsjkurs i Python for realfagslærere

Del 2: Funksjoner, plotting og numerisk løsning av likninger

kodeskolen simula

kodeskolen@simula.no

Sist uke gikk vi igjennom en grunnleggende innføring i Python-programmering. Denne uka skal vi fortsette med to programmeringskonsepter som er svært relevante for realfag, nemlig funksjoner og plotting av funksjoner. Kurset avsluttes ved at vi foreslår et konkret prosjekt man kan kjøre i klasserommet som handler om å løse likninger numerisk med *halveringsmetoden*

Det er mange grunner til at det er nyttig å løse likninger numerisk. For det første er det ikke sikkert at det er mulig å løse likningen analytisk ($cos(x) - x = 0$ er et eksempel på en slik likning), til tross for det er det null problem for en numerisk metode å finne en veldig god tilnærming. I tillegg til dette er det sjeldent man har behov for det korrekte svaret, et svar som er 10^{-10} unna er godt nok.

Å bruke programmering til å løse likninger er også svært relevant for noen kompetansemål i den nye 1T læreplanen:

- Formulere og løyse problem ved hjelp av algoritmisk tenking, ulike problemløysingsstrategiar, digitale verktøy og programmering
- Utforske strategiar for å løyse likningar, likningssystem og ulikskapar og argumentere for tenkjemåtane sine

Innhold

1 Funksjoner

Det første vi skal dekke idag er *funksjoner*. Vi har så langt allerede sett, og brukt, funksjoner i Python. For eksempel er `print`, `input`, og `sqrt` alle eksempler på funksjoner som vi bruker.

Felles for funksjoner er at vi kan bruke dem ved å *kalle* på dem, og at de da utfører en forhåndsdefinert oppgave. For eksempel kan vi kalle på `print`-funksjonen for å skrive ut en beskjed, vi kan kalle på `input`-funksjonen for å stille brukeren et spørsmål og vi kan kalle på `sqrt`-funksjonen for å regne ut kvadratroten av et tall.

Når vi kaller på en funksjon bruker vi alltid vanlig, runde, parenteser, `()`, etter funksjonsnavnet. Om funksjonen skal ta noe input, som de ofte gjør, skrives dette inn i parentesene. For eksempel kan vi regne ut $\sqrt{4}$ ved å skrive `sqrt(4)`.

1.1 Definere egne funksjoner

I tilegg til å bruke de innebygde funksjonene i Python, kan vi lage våre egne. Dette kaller vi å *definere* en funksjon, og derfor bruker vi nøkkelordet `def` for å gjøre dette, som er kort for "define", eller "definer" på norsk.

La oss se på et eksempel:

```
1 def f(x):
2     return x**2 + 2*x + 1
```

Her skriver vi først `def`, fordi vi ønsker å definere en funksjon, deretter skriver vi navnet vi ønsker for funksjonen vår, i dette tilfellet velger vi `f`. Deretter skriver vi parenteser og skriver inn hva slags *input* funksjonen ønsker. I dette tilfellet ønsker vi én ukjent, og velger og kalle denne for x. De "ukjente"variablene til funksjonen kalles gjerne for et *funksjonsargument*, eller bare et argument. Til slutt trenger vi et kolon, slik som for løkker og betingelser.

Etter linjen som starter med `def` kommer innholdet til selve funksjonen, og her trenger vi et innrykk. Alle kodelinjer med innrykk etter definisjonslinja hører til funksjonen. Disse kodelinjene utføres når funksjonen kalles. Det som i skjer i dette tilfellet, er at vi regner ut en tallverdi basert på den ukjente variabelen x, og så *returnerer* vi denne.

Å *returnere* betyr å sende tilbake ut av funksjonen. Vi kan altså tenke på det som

returneres som *resultatet* av funksjonskallet.

Som du kanskje har forstått er funksjonen vi her har definert, en matematisk annengradsligning. Vi kunne like gjerne ha skrevet den slik:

$$f(x) = x^2 + 2x + 1.$$

Når vi kjører kodesnutten over, så *definerer* vi funksjonen, det betyr at etter disse kodelinjene er kjørt, så eksisterer `f` som en funksjon vi kan bruke fritt senere i programmet vårt. Men det skjer ingenting mer når vi definerer funksjonen. Dersom vi ønsker å faktisk bruke funksjonen, eller å få noe *resultat* ut av den, må vi *kalle* på den.

Vi kan her for eksempel kalle på funksjonen ved å skrive `f(4)`, og da kjøres kodelinjene inne i funksjonen med $x = 4$, la oss teste dette selv:

```
1 print(f(4))
```

```
25
```

Om vi da regner ut for hånd for å sammenligne får vi

$$f(4) = 4^2 + 2 \cdot 4 + 1 = 16 + 8 + 1 = 25,$$

som er det vi forventet å få.

Vi kan også lagre resultatet fra et funksjonskall til en variabel, istedenfor å skrive det rett ut

```
1 resultat = f(2)
```

Når vi skriver denne koden, så definerer vi en ny variabel, og det er returverdien fra funksjonen, som lagres i variabelen vi definerer.

1.1.1 Funksjoner for å unngå å repetere

En grunn til å definere en funksjon, er om vi oppdager at vi gjentar en beregning eller prosess mange ganger. Da kan vi heller definere denne beregningen eller prosessen én gang, og så heller bare kalle på en funksjon som gjør jobben for oss etter det. På den måten unngår vi unødvendig repetisjon.

Ta for eksempel en volumberegning av en kule. Si vi ønsker å sammenligne størrelsene på baller brukt i mange forskjellige sporter. Da kunne vi skrevet følgende kode:

```
radius_basketball = 12
radius_fotball = 11
radius_håndball = 9
radius_tennisball = 3.3
radius_golfball = 2.3

volum_basketball = 4*pi*radius_basketball**3/3
volum_fotball = 4*pi*radius_fotball**3/3
volum_håndball = 4*pi*radius_håndball**3/3
volum_tennisball = 4*pi*radius_tennisball**3/3
volum_golfball = 4*pi*radius_golfball**3/3
```

Denne koden fungerer helt fint, men istedenfor å gjenta formelen for hver eneste beregning, så kunne vi definert den som en funksjon en gang først, og så brukt funksjonen flere ganger:

```
def kulevolum(radius):
    return 4*pi*radius**3/3

volum_basketball = kulevolum(radius_basketball)
volum_fotball = kulevolum(radius_fotball)
...
```

Dette gjør koden både lettere å lese og lettere å forstå. Samtidig er det enklere for oss å gå inn og fikse funksjonen om det skulle vise seg å være noe galt med den. Da trenger vi bare å fikse på ett sted i koden.

1.1.2 Navngiving av funksjoner og lesbarhet

Som vi nettopp, kan funksjoner hjelpe oss til å skrive kode som er lettere å forstå. Det er viktig fordi vi ønsker ofte at andre mennesker skal forstå hva vi har kodet, ikke bare datamaskinen. Vi ønsker også å være sikre på at vi forstår koden vi skriver selv. Veldig ofte når vi får en feil, kommer det av at koden ikke gjør det vi forventer. Og det kommer ofte av at vi ikke helt har forstått koden selv. Så en viktig evne å øve på når man lærer seg å programmere er å skrive forståelig kode.

Når du koder en funksjon er det derfor viktig at du tenker litt igjennom hva

funksjonen skal hete. I eksempelet over har vi valgt å kalle funksjonen "kulevolum". "kulevolum". er et bra navn fordi det forteller hva funksjonen gjør, den regner ut volumet av en kule. Dette gjør linje 4 og 5 i eksempelet over lette og forstå. `volum_basketball` blir definert som `kulevolum` for en kule med radiusen til en basketball. Tenk deg dersom funksjonen istedet hadde hatt navnet `v(r)`, eller enda værre `f(x)`. Da ville koden blitt mye vanskeligere å tolke.

1.1.3 Flere input

Det er fullt mulig å lage en funksjon som tar mer enn én input. La oss for eksempel si vi ønsker å lage en funksjon som bruker Pytagoras' regel for å regne ut en hypotenus, da kan vi definere den som følger:

```
1 from math import sqrt
2
3 def hypotenus(a, b):
4     return sqrt(a**2 + b**2)
```

Merk at vi må importere `sqrt` for at funksjonen skal fungere, men dette trenger vi ikke å gjøre inne i funksjonen, det holder å gjøre det i toppen av programmet vårt.

1.1.4 Lengre funksjoner

Funksjoner kan inneholde mer enn en linje kode. Vi kan ha mer kompliserte funksjoner som gjør mer sammensatte ting. Her er et eksempel på en funksjon som tar inn to tall og returnerer det tallet som er størst. For å få til det brukes en `if`/`else`-betingelse inne i funksjonen.

```
1 def maks(a, b):
2     if a > b:
3         størst = a
4     else:
5         størst = b
6     return størst
```

1.1.5 Flere output

Vi kan også fint lage en funksjon som returnerer mer enn én ting, da legger vi bare det vi ønsker å returnere etter hverandre med komma mellom. Her er et eksempel på en funksjon som tar inn en teller og en nevner for en brøk og returner både heltall og rest for brøken.

```
def divisjon(teller, nevner):
    heltall = teller//nevner
    rest = teller % nevner
    return heltall, rest

teller = 9
nevner = 5
heltall, rest = divisjon(teller, nevner)
print(f'{a}/{b} = {heltall} + {rest}/{nevner}')
```

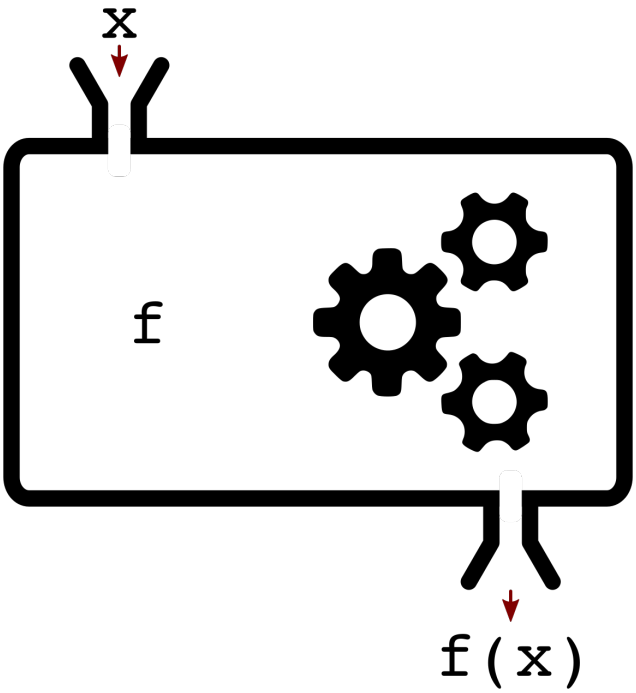
```
9/5 = 1 + 4/5
```

1.2 Analogier for funksjoner

Funksjoner kan være et litt vanskelig konsept å forstå, og det kan være verdt og bruke tid på å forklare hva funksjoner egentlig *er*. For noen elever er funksjoner velkjente fra matematikken, og isåfall handler det mest om å generalisere konseptet til programmering. For andre er kanskje funksjoner i matematikken noe vanskelig og forvirrende, og isåfall kan det kanskje hjelpe og snakke om og bruke funksjoner i programmeringssammenheng. Uansett grunn kan det være nyttig å ha noen analogier for hva funksjoner er og gjør.

Kanskje den enkleste forklaringen på en funksjon er at det er en *regel*. Når vi definerer en funksjon, så lager vi en regel for hva som skje når vi bruker den. Her kan det for eksempel være nyttig å implementere funksjoner frakoblet fra datamaskin, og vi kommer til å dekke dette i neste seksjon.

En noe mer komplisert analogi som ofte brukes, er å tenke på en funksjon som en slags *maskin*. Denne maskinen har en åpning på den ene side, hvor vi kan mate noe inn. Når vi putter noe inn i maskinen, for eksempel et tall, begynner den å jobbe. Etter maskinen har blitt ferdig å jobbe, spytter den så ut resultatet på den andre siden. Denne maskinen er funksjonen vår. Selve maskinen har et navn, i



dette tilfellet `f`. Vi kan nå mate inn noe i maskinen, her markert som en `x`, og vi får resultatet ut.

I programmeringsverden kaller vi gjerne en slik maskin for en "black box". Med det mener vi at selve innholdet i maskineriet er skjult og bygget inn. Det betyr at når man bruker en funksjon, så trenger vi ikke å tenke på hva innholdet i selve funksjonen/maskinen er, vi trenger bare å tenke på hva den egentlig gjør med inputen.

Tenk for eksempel på en brødbakemaskin. Instruksene sier hva du må putte inn i maskinen, og så får du et ferdig bakt brød ut. For å bruke denne maskinen trenger du ikke å vite hvordan den er skrudd sammen, eller hvordan den fungerer.

Det at funksjoner fungerer som svarte bokser, eller "black boxes" viser noe av styrken ved funksjoner. Ved å bare måtte tenke på de kompliserte detaljene når man lager funksjonen, og ikke når man bruker dem, så blir det en *abstraksjon* som hjelper oss bryte ned store og kompliserte problemer i enklere biter.

1.3 Frakoblede funksjoner

For å få en bedre forståelse for hva funksjoner er, og hvordan de fungerer, kan man jobbe med funksjoner frakoblet fra datamaskin, altså som en "unplugged" aktivitet. Her kan man for eksempel leke en enkel lek med to spillere, der man skal prøve å gjette hvilken funksjon den andre spilleren har implementert. Dette illustrerer også konseptet "black box".

Sett elever sammen i par. En av spillerene får rollen som funksjon, denne eleven får så enten utdelt en funksjon fra læreren, eller får lov til å skrive sin egen funksjon. Den andre spilleren skal nå gi input til funksjonen. For hver input som gis må spilleren som har funksjonsrollen gå igjennom stegene i funksjonen og gi en verdi i retur.

Etter å ha gitt litt forskjellig verdier som input, og fått litt forskjellig verdier i retur, kan spilleren nå begynne å gjette seg frem til hva funksjonen er. Etter hver input-output som gis, får spilleren lov til å gjette på en funksjonsdefinisjon. Her bør man ikke gjette blindt, da blir det helt umulig å treffe. Isteden bør de forklare hvorfor de gjetter det de gjør, basert på inputen og outputen de har fått. Fortsett å gi input og output til spilleren klarer å gjette funksjonen, i hvilket tilfelle runden er over og de to spillerene bytter roller. For å gjøre litt konkurranse ut av det kan man si at det er om å gjøre å gjette riktig på så få input/output som mulig.

Før man leker denne leken bør man ha satt noen klare rammer på hva slags funksjoner man kan lage, og vanskelighetsnivået. Alternativ kan læreren lage funksjonskort på forhånd, som deles ut tilfeldig til elevene.

Eksempler på enkle funksjoner:

```
def f(x):
    return x + 4
```

```
def f(x):
    return 2*x
```

```
def f(x):
    return x - 10
```

Eksempler på vanskeligere funksjoner:

```
def f(x):
    return 2*(x - 1)
```

```
def f(x):
    return x**2
```

Etterhvert som elevene lærer mer, kan man begynne å lage funksjoner som bruker mer av det man kan om Python. For eksempel en funksjon som tar inn to ting

```
def f(x, y):
    return 2*x + y
```

Eller kanskje en løkke som også inneholder en `if`-betingelse:

```
def f(x, y):
    if x > y:
        return x
    else:
        return y
```

Slike funksjoner kan fort bli såpass kompliserte at de blir vanskelig å gjette seg frem til, så her er enten gode retningslinjer, eller ferdiglagde funksjonskort gode valg.

1.4 Funksjoner i programmering – mer enn bare matematikk

Alle eksemplene vi har brukt så langt har vært matematiske funksjoner. I matematikk er funksjoner som oftes noe som regner på tall. I programmering derimot, kan funksjoner gjøre nesten hva som helst. Dette er fordi funksjoner kan inneholde hva som helst av kode og det betyr at de kan gjøre alt som vi ellers gjør i programmene våre.

For eksempel kan vi lage en funksjon som skriver ut en beskjed til en person:

```
def hils_på(navn):
    print(f'Hei på deg {navn}!')

hils_på('Pål')
hils_på('Kari')
```

```
Hei på deg Pål!
Hei på deg Kari!
```

Merk at funksjonen her inneholder `print`-kommandoen, og skriver derfor rett ut til skjermen når vi bruker den. Dette er ulikt fra våre tidligere eksempler, hvor vi isteden brukte `return` for å *returnere* en tallverdi, som vi så skrev ut utenfor funksjonen.

Som du ser i eksempelet over er det fullt mulig å lage en funksjon uten en returverdi. Men da må vi passe på litt når vi bruker den. Dersom vi for eksempel prøver å definere en variabel på denne måten, kan vi bli litt overrasket over resultatet:

```
1 resultat = hils_på('Marius')
2 print(resultat)
```

```
Hei på deg Marius
None
```

Her ser vi at det første som skjer er at selve beskjeden skrives ut, for det er jo dette vi har definert at funksjonen skal gjøre. Når vi skriver ut variabelen `resultat` derimot, får vi bare `None` ut. Dette er fordi vi ikke har noen returverdi, og variabelen skal jo settes til returverdien. Isteden blir den satt til `None`, altså "ingenting".

2 Plotting av funksjoner

Å tegne, eller plotte, funksjonsgrafer er utrolig nyttig i matematikken. Det som er praktisk med Python er at det har et av de kraftigste plotteverktøyene som eksisterer – *matplotlib*. I denne guiden skal vi lære hvordan vi kan tegne enkle grafer i matplotlib gjennom det hendige *pylab*[1] biblioteket. Før vi kan begynne på dette må vi introdusere en ny variabeltype – arrays.

[1] Pylab er en samling av verktøy fra tre vitenskapelige python bilbiotek; matplotlib, NumPy og SciPy

2.1 Arrays

En *array* er en liste med tall som vi kan manipulere effektivt med matematiske funksjoner. La oss ta et eksempel hvor vi ønsker å regne ut x+1 for alle heltall mellom en og ti. Da kan vi skrive den følgende koden:

```
1 from pylab import arange
2
3 x = arange(1, 11)
4 y = x + 1
5 print(x)
6 print(y)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
[ 2  3  4  5  6  7  8  9 10 11]
```

Hva er det som skjer her? I den første linja så importerer vi arange fra pylab. Dette er en funksjon som minner veldig om range. Forskjellen er at range gir oss tallrekker vi kan bruke i en `for` løkke, men arange gir oss tallrekker som arrays. I linje 3 så opretter vi en array `x`, som inneholder alle tall fra og med 1 til, men ikke med, 11. Deretter, legger vi 1 til `x` og lagrer resultatet i `y`. Når vi printer dette ser vi at en ble lagt til i hvert eneste element i x, akkurat slik vi ville.

Fordelen med å bruke arrays er altså at vi kan utføre matematiske operasjoner på en hel tallrekke samtidig. Dette er nyttig av flere årsaker; for det første, jobber man ofte med tallrekker i matematikk. I tillegg til dette, er arrays det som brukes når vi skal lage plot i Python.

2.1.1 Bruke matematiske operasjoner på arrays

I eksempelet over la vi til 1 til en array med elementer fra 1 til 10. Dersom vi legger sammen en array med et tall, vil tallet bli lagt til hvert element i arrayen. På samme måte kan vi bruke gange, dele eller opphøye en array med et tall.

```
1 from pylab import arange
2
3 x = arange(1, 11)
4
5 produkt = x * 2
6 divisjon = x / 2
```

```
potens = x**2

print(f'x: {x}')
print(f'x*2: {produkt}')
print(f'x/2: {divisjon}')
print(f'x**2: {potens}')
```

```
x: [ 1  2  3  4  5  6  7  8  9 10]
x*2: [ 2  4  6  8 10 12 14 16 18 20]
x/2: [0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5. ]
x**2: [  1   4   9  16  25  36  49  64  81 100]
```

Vi kan også gjøre andre, enkle matematiske oprasjoner med to arrays. Vi kan for eksempel legge sammen to arrays, da vil hvert element av hver array legges sammen:

```
from pylab import arange

x = arange(1, 11)
y = arange(2, 12)

z = x + y
print(f'x: {x}')
print(f'y: {y}')
print(f'x + y: {z}')
```

```
x: [ 1  2  3  4  5  6  7  8  9 10]
y: [ 2  3  4  5  6  7  8  9 10 11]
x + y: [ 3  5  7  9 11 13 15 17 19 21]
```

Vi kan også bruke multiplikasjon og divisjon:

```
from pylab import arange

x = arange(1, 6)
y = arange(2, 7)

print(f'x: {x}')
print(f'y: {y}')
print(f'x * y: {x * y}')
print(f'x / y: {x / y}')
```

```
x: [1 2 3 4 5]
y: [2 3 4 5 6]
x * y: [ 2  6 12 20 30]
x / y: [0.5        0.66666667 0.75       0.8
   0.83333333]
```

En ting som er viktig å huske på, er arrayene må ha samme størrelse. Hvis vi for eksempel prøver å legge sammen to arrayer med ulik størrelse for vi en feilmelding:

```
from pylab import arange

x = arange(1, 10)
y = arange(2, 12)

z = x + y
```

```
ValueError: operands could not be broadcast together with
   shapes (9,) (10,)
```

Dette er en litt mystisk feilmelding, men den forteller oss at det ikke går ann å legge sammen to rekker av ulik lengde.

2.1.2 Bruke funksjoner på arrays

Vi har tidligere sett hvordan vi kan hente avanserte matematiske funksjoner fra `math` pakken. Et naturlig spørsmål er om disse funksjonene også kan brukes på array-er. Svaret på dette er dessverre nei, men det eksisterer en kopi av alle `math` funksjonene i `pylab` pakken. Denne varianten av funksjonene virker både på arrayer og på vanlige tall, som demonstrert med den følgende koden

```
from pylab import arange, exp, sin

a = 0.5
b = exp(a)
c = sin(a)

print(f'a: {a}')
print(f'exp(a): {b}')
print(f'sin(a): {c}')
```

```
10
11 x = arange(1, 6)
12 y = exp(x)
13 z = sin(x)
14
15 print(f'x: {x}')
16 print(f'exp(x): {y}')
17 print(f'sin(x): {z}')
```

```
a: 0.5
exp(a): 1.6487212707001282
sin(a): 0.479425538604203
x: [1 2 3 4 5]
exp(x): [  2.71828183   7.3890561   20.08553692
   54.59815003 148.4131591 ]
sin(x): [ 0.84147098  0.90929743  0.14112001 -0.7568025
   -0.95892427]
```

Her ser vi altså at `pylab` sin eksponentialfunksjon og sinusfunksjon virker både med vanlige tall og arrayer. En oversikt over nyttige matematiske funksjoner som finnes i `pylab` er gitt i Tabell 1 .

Matematisk funksjon	Pylab funksjon
$\sqrt{x}$	`sqrt(x)`
e^x	`exp(x)`
$sin(x)$	`sin(x)`
$cos(x)$	`cos(x)`

Tabell 1: Noen nyttige matematiske funksjoner i Pylab

Vi har nå sett hvordan vi kan bruke `pylab` sine funksjoner på arrayer, vi har og lært hvordan vi lager våre egne funksjoner. Et naturlig spørsmål å stille oss selv er derfor: kan vi definere funksjoner som virker med arrays?". Svaret på dette er ja. La oss ta utgangspunkt i det første eksempelet vårt med arrays, hvor vi la 1 til alle elementene i en array. Om vi ønsker en funksjon som gjør dette kan vi skrive den følgende koden

```
1 from pylab import arange
2
3 def legg_til_en(tall_eller_array):
```

```
    return tall_eller_array + 1

x = arange(1, 11)
y = legg_til_en(x)

print(x)
print(y)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
[ 2  3  4  5  6  7  8  9 10 11]
```

Her har vi altså en funksjon som virker både for vanlige tall og arrays! La oss gå et steg videre, og se på `maks` funksjonen vi definerte tidligere.

```
from pylab import arange

def maks(a, b):
    if a > b:
        størst = a
    else:
        størst = b
    return størst

x = arange(1, 11)
y = x + 1
z = max(x, y)
print(x)
print(y)
print(z)
```

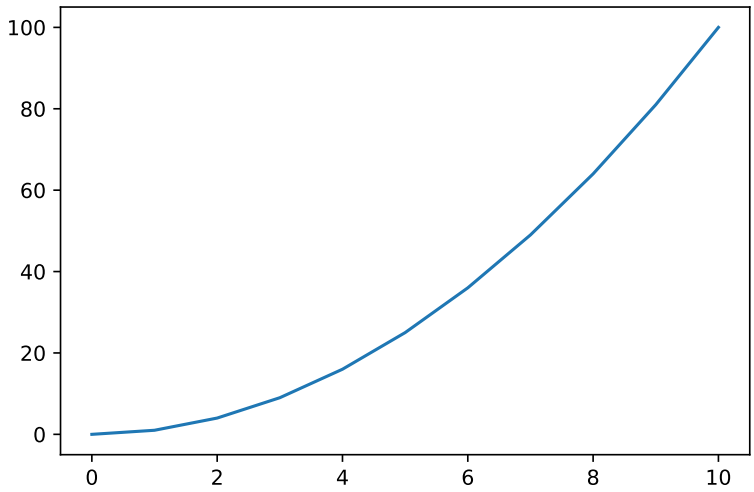
```
ValueError: The truth value of an array with more than one
    element is ambiguous. Use a.any() or a.all()
```

Nok en gang får vi en litt kryptisk feilmelding. Det den klager over her er selve `if`-betingelsen. Vi prøver å sjekke om `a` er større enn `b`, men både `a` og `b` er tallrekker. Python vet ikke hvordan den skal sammenlikne to tallrekker – ønsker vi å sammenlikne de minste tallene i `a` og `b`, de største tallene, eller kanskje hvor mange tall som er i hver tallrekke? Siden Python ikke vet dette får vi altså en feilmelding.

2.2 Å tegne grafer i Python

Nå som vi vet hva array-er er og hvordan de fungerer kan vi endelig starte å tegne funksjonsgrafer. La oss begynne med et eksempel

```
1 from pylab import arange, plot, show
2
3 x = arange(0, 11)
4 y = x**2
5
6 plot(x, y)
7 show()
```



La oss se hva som skjer her. I første linjen så importerer vi `arange`, som vanlig, men her har det dukket opp to nye funksjoner; `plot` og `show`. `plot` er den funksjonen vi bruker for å tegne en graf, og `show` er den funksjonen vi bruker for å vise frem grafene vi har tegnet.

Videre, i linje tre oppretter vi en array `x`, som inneholder alle tall fra og med 0 til og med 10. Dette er x-koordinatene våre. I linje fire oppretter vi en variabel `y`, som inneholder kvadratet av x-koordinatene våre. `y` representerer y-koordinatene våre.

Etter at vi har opprettet x og y koordinatene våre, bruker vi `plot`-funksjonen vi

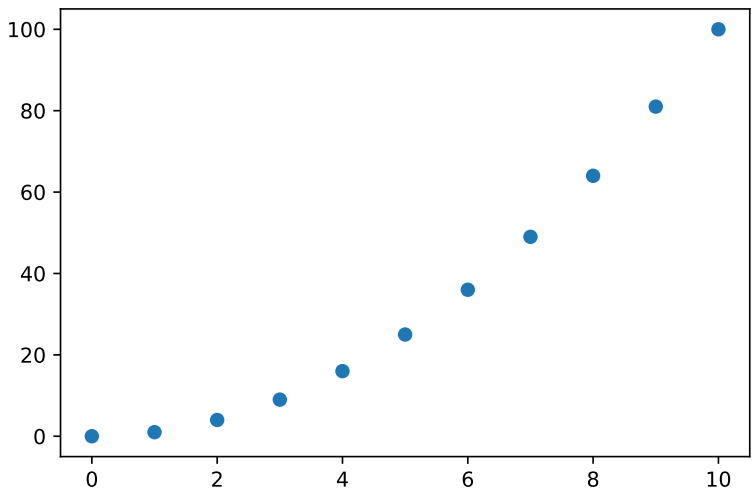
importerte fra `pylab`. `plot` tar inn en array med x koordinater og en array med y koorindater, plasserer et punkt i hvert x-y koordinat og tegner streker mellom disse punktene. Til sist må vi bruke `show` funksjonen for å vise frem plottet vi lagde.

Denne måten å lage plott på minner om hvordan vi gjør det for hånd på ruteark. Først må vi bestemme x-koordinater og regne ut tilhørende y-koordinater. Så tegner vi opp punktene for hvert par med koordinater og tilslutt trekker vi en strek igjennom.

For å gjøre det tydeligere kan vi også be python om å tegne kun punktene, uten streker. Det gjør vi med å senne et ekstra argument, `'o'`, til `plot`:

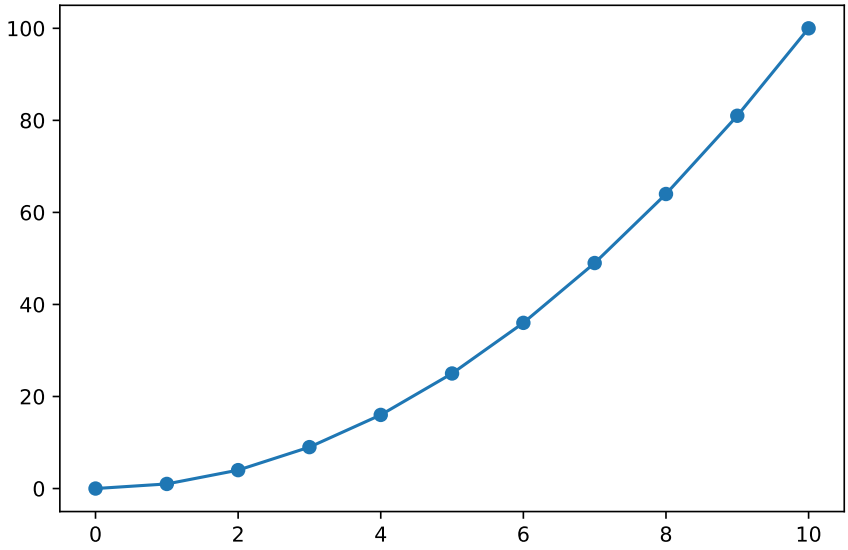
```
from pylab import arange, plot, show

x = arange(0, 11)
y = x**2

print(f'x: {x}')
print(f'y: {y}')
plot(x, y, 'o')
show()
```

```
x: [ 0  1  2  3  4  5  6  7  8  9 10]
y: [  0   1   4   9  16  25  36  49  64  81 100]
```



Her ser vi altså at en sirkel ble plassert i koordinatene $(0, 0)$, en ble plassert i $(1, 1)$, en i $(2, 4)$ osv. De samme koordinatene vi ser når vi tegner grafer for hånd.

Det er også mulog å kobinere de to siste plottene vi har lagd og dermed ha et plot hvor strekene er tegnet med sirkler i hvert punkt. For å gjøre det bytter vi ut `'o'` med `'o-'` i koden over. Prøv dette selv og sammenlikn med figuren under.



2.3 Plottestil

Det vi akkurat demonstrerte er forskjellige tre forskjellige plottestiler. Nedenfor er en tabell med noen flere slike stiler.

Plottestil	Tekststreng
Vanlig strek	`'-'`
Stiplet strek	`'--'`
Sirkel	`'o'`
Trekant	`'^'`
Stjerne	`'*'`
Strek med sirkel	`'-o'`

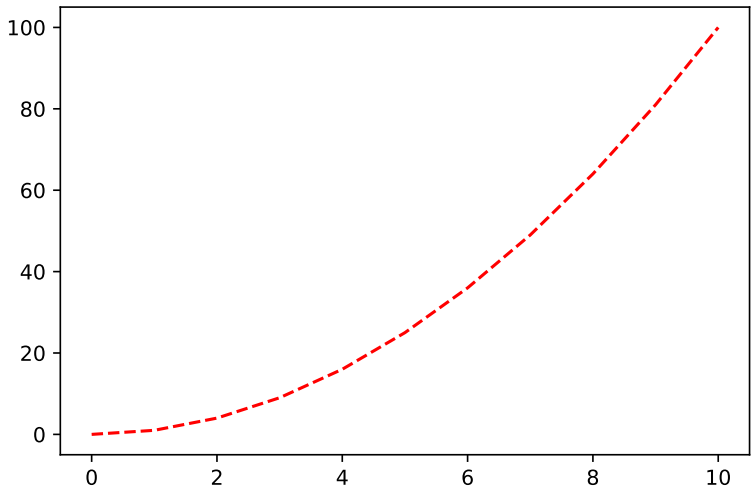
Tabell 2: Oversikt over plottestil-tekststrenger

Farge	bokstav
Grøn	`g`
Gul	`y`
Magenta	`m`
Blå	`b`
Lyseblå	`c`
Rød	`r`

Tabell 3: Oversikt over farger

Plot tar altså tre argument, en array med x-koordinatene, en array med y-koordinatene og en tekststreng for å velge plottestil.

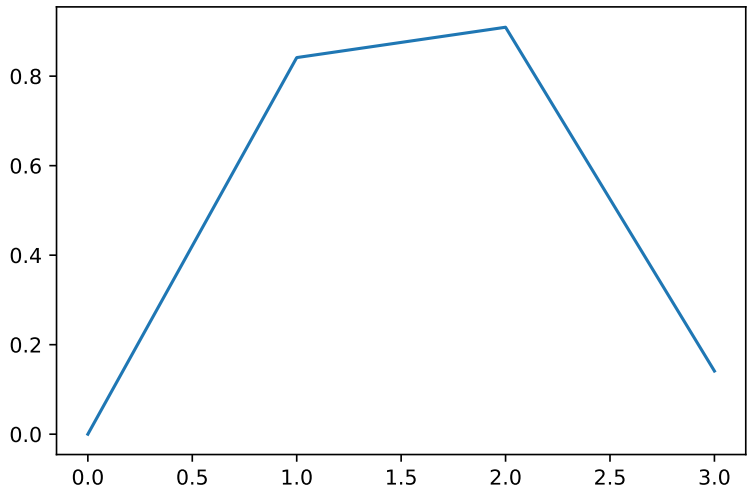
Det går også an å endre farge på plottet. Det gjøres ved at du legger en bokstav som indikerer farge på slutten av plottestil-tekststrengen. Hvis du for eksempel vil plotte `x` og `y` med en *rød*, stiplet linje kan du gjøre det med kommandoen `plot(x, y, '--r')`. Den første biten av tekststrengen (`--`) forteller Python at du ønsker en stiplet linje, og den siste bokstaven (`r`)står for **r**ed eller rød. Nedenfor er en figur som illustrerer akkurat dette. Tabell 3 viser en oversikt over hvilke bokstaver som gir hvilke farger.



2.3.1 Tettere punkter

I alle eksemplene vi har sett på hittil har x-koordinatene vært $1, 2, 3, ...$ Dette fungerer ofte fint, men av og til ønsker vi tettere x-koordinater. La oss si at vi ønsker å plotte en sinus funksjon fra 0 til 3. Dette kan vi gjøre slik:

```
from pylab import sin
x = arange(0, 4)
y = sin(x)

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



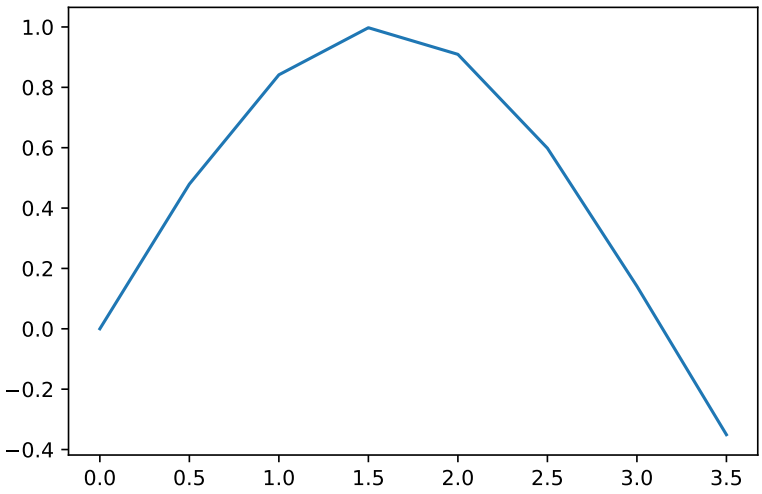
Som vi ser ble ikke dette en veldig pen sinus kurve. Det kommer av at vi plotter for punktene (0, sin(0)), (1, sin(1), (2, sin(2) og (3, sin(3)) , hvilket er ganske få punkter med mye mellomrom. Heldigvis så er det en måte vi kan få tettere punkt på. `arange` funksjonen har nemlig et argument som bestemmer mellomrommet mellom hvert element i arrayet. La oss demonstrere dette med et eksempel.

```
from pylab import arange

x = arange(0, 4, 0.5)
print(f'x: {x}')
```

```
[0.   0.5 1.   1.5 2.   2.5 3.   3.5]
```

Her ser vi altså at vi får en array med en rekke med tall som starter på null. Mellom hvert tall er det et mellomrom på 0.5 og alle tall i tallrekka er mindre enn 4. La oss plotte en sinus funksjon hvor avstanden mellom x-verdiene er 0.5

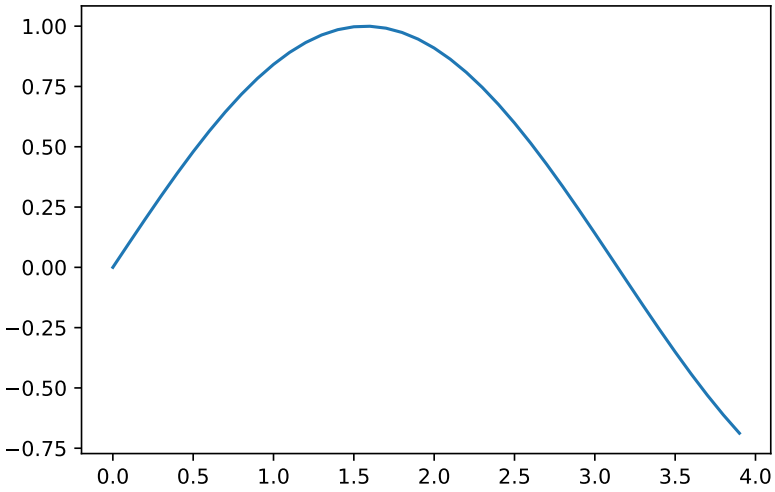
```
from pylab import arange, sin, plot, show

x = arange(0, 4, 0.5)
y = sin(x)

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



Her ser vi at oppløsningen ikke var helt god nok, La oss derfor prøve med en oppløsning på 0.1

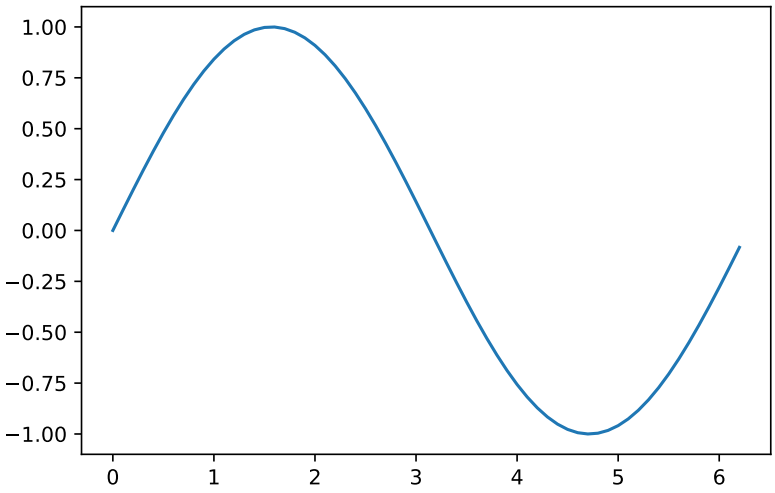
```
from pylab import arange, sin, plot, show

x = arange(0, 4, 0.1)
y = sin(x)

plot(x, y)
```

```
7 show()
```



Her fikk vi absolutt en finere sinuskurve. Det eneste problemet er at sinuskurven vår ikke slutter i $x = 2\pi$. I motsetning til `range` kan `arange` slutte på et desimaltall, så vi kan velge x-koordinater som går fra 0 til π. For å få til dette er det nok at vi importerer `pi` fra `pylab` (eller `math`) og bruker `arange(0, 2*pi +0.1, 0.1)` slik:

```
1 from pylab import arange, sin, plot, show, pi
2
3 x = arange(0, 2*pi + 0.1, 0.1)
4 y = sin(x)
5
6 plot(x, y)
7 show()
```



Nå har vi en fin sinus kurve!

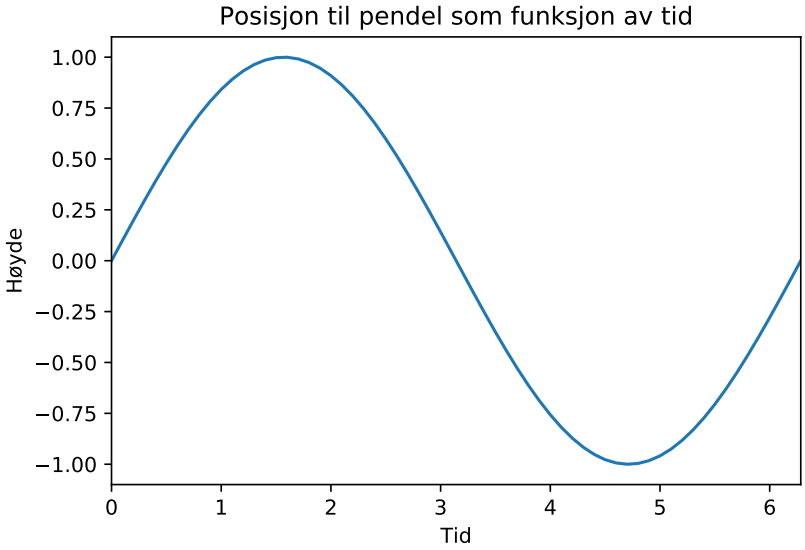
2.3.2 Pynte på plottet

I eksemplene over ser vi at "rammen" til plottet stikker litt utenfor graflinja vår. Dette kan vi fikse med å definere hvor grensene til x-aksen går med `xlim` funksjonen i `pylab`. `xlim` tar inn to tall som definerer hvor x-aksen skal starte og slutte.

Det er ofte lurt å legge til merkelapper på aksene som sier noe om hva aksene måler. Og en tittel øverst som beskriver plottet. Dette kan vi gjøre med henholdsvis `xlabel`, `ylabel` og `title` funksjonene. Her er et eksempel som kombinerer disse i et plot.

```
from pylab import sin, pi, plot, show, xlim, xlabel,
   ylabel, title
x = arange(0, 2*pi+0.1  , 0.1)
y = sin(x)

plot(x, y)
xlim(0, 2*pi)
xlabel('Tid')
ylabel('Høyde')
title('Posisjon til pendel som funksjon av tid')
```

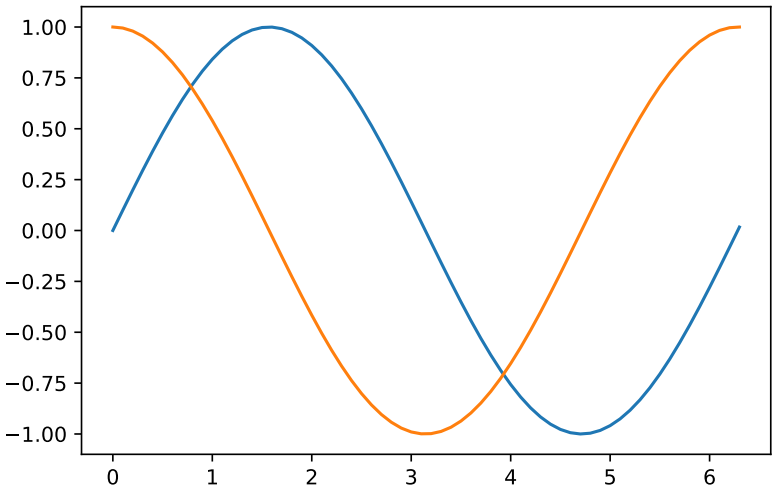
```
10 show()
```



2.3.3 Flere plot

Vi kan også plotte to grafer i samme plot. Da kaller vi bare `plot` en gang til før `show` for å tegne opp en ny graf i samme plottet.

```
1 from pylab import sin, cos, pi, plot, show
2 x = arange(0, 2*pi+0.1  , 0.1)
3 y1 = sin(x)
4 y2 = cos(x)
5
6 plot(x, y1) # første plot
7 plot(x, y2) # andre plot
8 show()
```

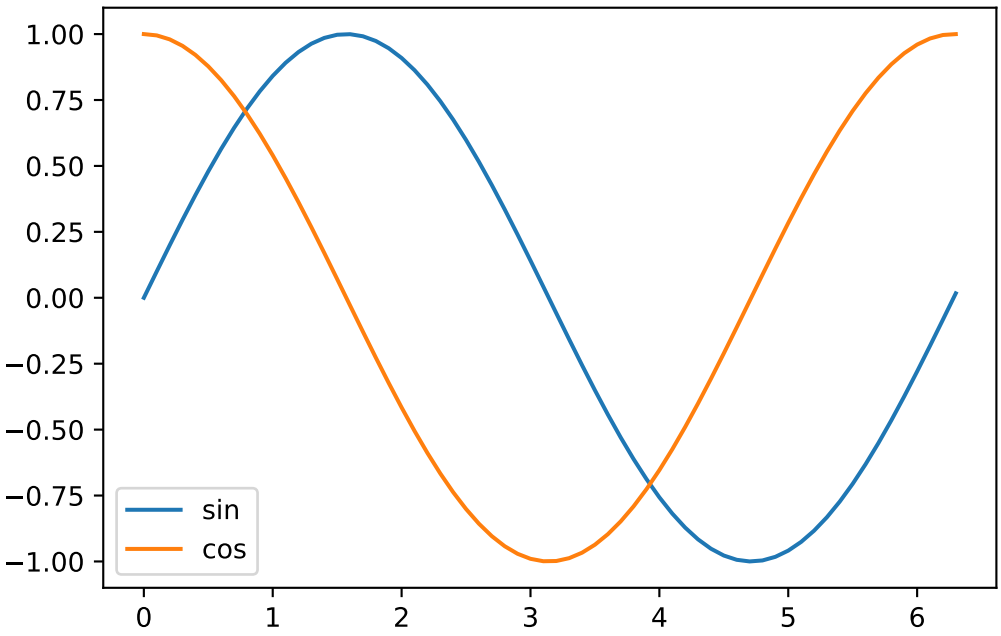


Når vi har to kurver i samme plottevindu kan det være lurt å signalisere hva de forskjellige plottene viser. Dette gjør vi med en *kurvemerkelapp*, eller på engelsk *legend*. Koden under viser hvordan legend funksjonen virker.

```
from pylab import sin, cos, pi, plot, show, legend
x = arange(0, 2*pi+0.1  , 0.1)
y1 = sin(x)
y2 = cos(x)

plot(x, y1, label='sin') # første plot
plot(x, y2, label='cos') # andre plot
legend()
show()
```

For å få kurvemerkelappen til å virke må vi bruke en funksjonalitet i Python vi ikke har snakket om før. Nemlig *keyword arguments*, eller nøkkelord-argument på norsk. Dette er en litt avansert funksjonalitet i Python, som lar oss endre hvordan funksjoner blir kalt. For dette eksempelet trenger vi bare å huske at vi må sende inn `label='sin'` for å fortelle at plottet skal ha merkelappen `'sin'`. Så må vi også kalle på `legend()` før `show()`. `legend()` funksjonen forteller python at vi ønsker å vise frem merkelappene.

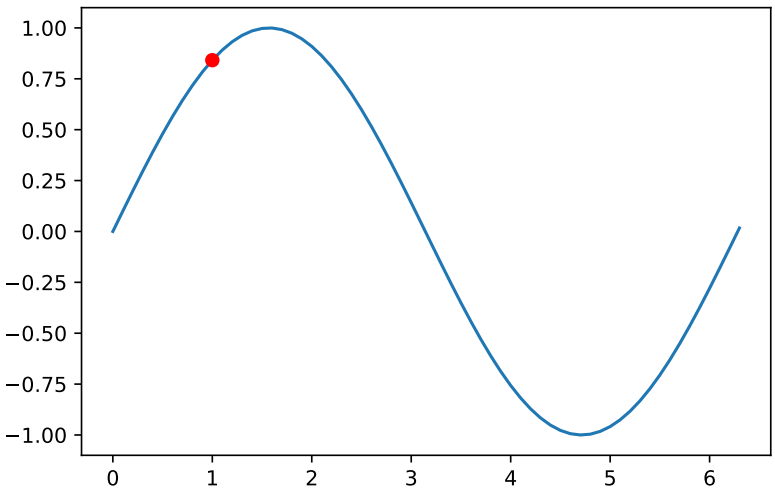
Hvis vi ønsker å markere et plot i punktet kan vi sende to tall in i `plot`. Her er et eksempel hvor vi ønsker å markere punktet $(1, \sin(1)$ med en rød sirkel.




```
from pylab import sin, plot, show, pi
x = arange(0, 2*pi+0.1  , 0.1)
y = sin(x)

plot(x, y)
plot(1, sin(1), 'ro') # rødt punkt
show()
```

Funksjonalitet	Pylab funksjon
Plotte	`plot(x,y)`
Vise frem plottet	`show()`
Tittel	`title('Din tittel')`
Merkelapp for x-aksen	`xlabel('Merkelapp for x-aksen')`
Merkelapp for y-aksen	`ylabel('Merkelapp for y-aksen')`
Sette grensene til x-aksen	`xlim(startverdi, stopverdi)`
Sette grensene til y-aksen	`ylim(startverdi, stopverdi)`
Vise grafmerkelapper	`legend()`

Tabell 4: Oversikt over plotte-funksjoner



Nå har vi gått igjennom ganske mange funkjsoner for plotting, så her har vi en tabell med en oversikt over de ulike funksjonene og hva de gjør.

3 Prosjekt: Løse likninger med datamaskinen

Her skal vi komme frem til en algoritme for å løse vilkårlige likninger ved hjelp av datamaskinen. Men, før vi begynner på det, skal vi fortsette med gjetteleken vi lærte om forrige uke.

3.1 Omvendt gjettelek

Sist uke lagde vi et program hvor datamaskinen tenker på et tall mellom `mintall` og `makstall`, og vi skal finne ut hva det er. Hver gang vi gjetter får vi vite om tallet er for høyt eller for lavt. Programmet vi endte opp med så slik ut:

```
fasitsvar = 45
maksforsøk = 100
print("Jeg har tenkt på et tall mellom 1 og 1000.")
print("Prøv å gjett det!")
print()

for forsøk in range(1, maksforsøk + 1):
    gjett = int(input("Gjett: "))

    if gjett == fasitsvar:
        print("Helt riktig!")
        print(f"Du brukte {forsøk} forsøk!")
        break
    elif gjett > fasitsvar:
        print(f"Ditt gjett på {gjett} er for høyt.")
    elif gjett < fasitsvar:
        print(f"Ditt gjett på {gjett} er for lavt.")

else:  # Hvis løkka ikke avsluttes med break
    print(f"Der har du prøvd {maksforsøk} ganger og gått
       tom for gjett! Riktig svar er {fasitsvar}")
```

I tillegg til dette, fant vi ut at den beste måten å gå frem på, var å gjette i midten av intervallet (`mintall`, `makstall`) og oppdatere det ut i fra hvilket svar vi får. Hvis du ikke husker dette, kan det være en idé å lese kapittel 5 i kompendiet til forrige kursdag. Dette kan vi bruke til å kode et programm hvor det i stedet er vi som tenker på et tall og datamaskinen som gjetter.

Det første vi trenger er en funksjon som returnerer et gjett. Og vi ønsker at gjettet skal være midtpunktet mellom det minste mulige tallet og det største mulige tallet. Vi trenger altså en funksjon som tar inn to tall og returner midtpunktet:

```
def midpunkt(a, b):
    return (a + b)/2
```

Vi vil altså ha en løkke hvor datamaskinen gjetter hvilket tall vi tenker på. Tilsvarende slik vi hadde det da vi hadde en løkke hvor vi gjetter hvilket tall datamaskinen tenker på.

```
maks_tall = 100
min_tall = 0

maks_forsøk = 100

print(f'Tenk på et tall mellom {min_tall} og {maks_tall}')
for forsøk in range(1, maks_forsøk+1):
    gjett = int(midpunkt(min_tall, maks_tall))
    print(f'Tenker du på {gjett}?')
```

```
Tenk på et tall mellom 0 og 100
Tenker du på 50?
Tenker du på 50?
Tenker du på 50?
...
Tenker du på 50?
Tenker du på 50?
```

Det observante øyet ser det står `gjett =int(midtpunkt(min_tall, maks_tall) )`. Grunnen til at vi bruker `int` er at vi tenker kun på heltall, så derfor vil vi at maskinen bare skal gjette heltal. Hvis datamaskinen gjetter et desimaltall så rundes det ned til nermeste heltall.

Vi ser også at datamaskinen alltid gjetter 50. For å oppdatere gjettet må jo vi gi en tilbakemelding om datamaskinen gjettet for høyt eller for lavt. Det kan vi gjøre med `input` funksjonen vi lærte forrige uke og en betingelse.

```
maks_tall = 100
min_tall = 0

maks_forsøk = 100

print(f'Tenk på et tall mellom {min_tall} og {maks_tall}')
for forsøk in range(1, maks_forsøk+1):
    gjett = int(midpunkt(min_tall, maks_tall))
    print(f'Tenker du på {gjett}?')
    svar = input('Var dette for høyt, for lavt eller
       riktig?')
```

```
    if svar == 'riktig':
        print(f'Jippi, jeg greide det på {forsøk} forsøk!')
        break
```

```
Tenk på et tall mellom 0 og 100
Tenker du på 50?

Var dette for høyt, for lavt eller riktig? riktig
Jippi, jeg greide det på 1 forsøk!
```

Det virker! Eller i hvertfall om datamaskinen gjetter riktig på første forsøk... La oss modifisere koden slik at den virker om vi sier for høyt eller for lavt og. Før vi gjør den modifiseringen må vi bestemme oss for hva datamaskinen skal gjøre om den gjetter for lavt eller for høyt.

Hvis 50 var for høyt så ønsker vi at datamskinen skal gjette mellom null og femti. For å gjøre det, oppdaterer vi `min_tall` og setter den lik `gjett` (som nå er 50). Tilsvarende, hvis 50 var for lavt oppdaterer vi `maks_tall` og setter den lik `gjett`. For å gjøre dette bruker vi `elif`-setningene vi lærte forrige uke.

```
maks_tall = 100
min_tall = 0

maks_forsøk = 100

print(f'Tenk på et tall mellom {min_tall} og {maks_tall}')
for forsøk in range(1, maks_forsøk+1):
    gjett = int(midpunkt(min_tall, maks_tall))
    print(f'Tenker du på {gjett}?')
    svar = input('Var dette for høyt, for lavt eller riktig? ')
    if svar == 'riktig':
        print(f'Jippi, jeg greide det på {forsøk} forsøk!')
        break
    elif svar == 'for høyt':
        maks_tall = gjett
    elif svar == 'for lavt':
        min_tall = gjett
    else:
```

```
19          print('Jeg forsto ikke hva du mente, du må svare "
                riktig", "for høyt" eller "for lavt"')
```

```
Tenk på et tall mellom 0 og 100
Tenker du på 50?

Var dette for høyt, for lavt eller riktig? for høyt
Tenker du på 25?

Var dette for høyt, for lavt eller riktig? for lavt
Tenker du på 12?

Var dette for høyt, for lavt eller riktig? riktig
Jippi, jeg greide det på 3 forsøk!
```

Her ser vi at datamaskinen krymper intervallet den gjetter i hver gang den gjetter feil helt til den til slutt gjetter riktig. Dette kalles *halveringsmetoden* eller *bisection method* på engelsk. Halveringsmetoden kan også brukes til å løse likninger numerisk.

3.1.1 Gjettelek for å gjette nullpunkt

La oss si at vi har en funksjon $f(x) = 5x - 125$ som vi vil løse. Vi kan koden denne funksjonen i Python slik

```
1 def f(x):
2     return 5*x - 125
```

Vi kan nå kode et program som gjør det å finne denne funksjonens nullpunkt til en gjettelek. For å gjøre det må vi bare gjøre bittesmå endringer på vårt originale gjettelek-program. I stedet for å sjekke om `gjett` er større enn, mindre enn eller lik `gjett`. Så sjekker vi om `f(gjett)` er større enn, mindre enn eller lik `0`.

```
1 def f(x):
2     return 5*x - 125
3
4 maks_tall = 100
5 min_tall = 0
6
7 maksforsøk = 100
```

```
print(f"Jeg har tenkt på en funksjon som har et nullpunkt mellom {min_tall} og {maks_tall}")
print("Prøv å gjett det!")
print()

for forsøk in range(1, maksforsøk + 1):
    gjett = float(input("Gjett: "))

    if f(gjett) == 0:
        print(f'Helt riktig, f({gjett}) er lik {f(gjett)}!')
        print(f'Du brukte {forsøk} forsøk!')
        break
    elif f(gjett) > 0:
        print(f'f({gjett}) er positiv.')
    elif f(gjett) < 0:
        print(f'f({gjett}) er negativ.')

else:  # Hvis løkka ikke avsluttes med break
    print(f'Der har du prøvd {maksforsøk} ganger og gått tom for gjett!')
```

```
Jeg har tenkt på en funksjon som har et nullpunkt mellom 0 og 100
Prøv å gjett det!


Gjett: 50
f(50) er positiv.

Gjett: 20
f(20) er negativ.

Gjett: 25
Helt riktig, f(25) er lik 0!
Du brukte 3 forsøk!
```

La oss prøve å endre funksjonen til noe som er litt vanskeligere, f.eks

$$f(x) = \frac{x}{100}^{5} - 0.1log\left(\frac{x+1}{100}\right) - 0.5 \tag{1}$$

Hvis vi bytter ut `f` funksjonen i programmet over med denne funksjonen, dukker det opp et problem. Det er nemlig ganske ofte at løsningstallet har mange desimaler. La oss se hva som skjer dersom vi gjetter et tall veldig nærme det nøyaktige nullpunktet.

```
Jeg har tenkt på en funksjon som har et nullpunkt mellom 0
    og 100
Prøv å gjett det!

Gjett: 86.588
f(86.588) = -1.6902841971366822e-05, som er negativ.
```

Vi ser at f(86.588) er veldig nærme 0. Det er har hele 4 nuller etter desimaltegnet. Likevel forteller maskinen oss at vi ikke har gjettet riktig.

Likninger er (som oftest) ikke noe vi løser for moroskyld. Vi løser dem som en del av et større problem, slik som "hvor høy spenning bør jeg ha i denne kretsen?", "hva er avstanden mellom disse hjernecellene?" eller "kommer kanonkula til å treffe borgen vår?". Da er vi ikke interessert i det "sanne" svaret med to streker under, for vi er en gang helt sikre på hvordan likningene vi løser skal se ut. Da er det nok å finne et omtrentlig svar.

Vi vil altså sjekke at tallet er nærme nok null. Det kan vi gjøre ved å sjekke at absoluttverdien til tallet er mindre enn en grense. Grensen setter vi etter hvor nøyaktig vi vil at svaret skal være. For å finne absoluttverdi kan vi bruke `abs` funksjonen som er innebygd i Python, eller lage vår egen etter forklaringene i oppgave 1 på slutten av kompendiet. Den ferdige koden blir da:

```
1  from pylab import log
2
3  def f(x):
4      return (x/100)**5 - 0.1 * log((x + 1)/100) - 0.5
5
6  maks_tall = 100
7  min_tall = 0
8
9  maksforsøk = 100
10 feilterskel = 0.01
11 print(f"Jeg har tenkt på en funksjon som har et nullpunkt
       mellom {min_tall} og {maks_tall}")
12 print("Prøv å gjett det!")
13 print()
```

```
14
15 for forsøk in range(1, maksforsøk + 1):
16     gjett = float(input("Gjett: "))
17
18     if abs(f(gjett)) > feilterskel:
19         print(f'Helt riktig, f({gjett}) er lik {f(gjett)}!
               ')
20         print(f'Du brukte {forsøk} forsøk!')
21         break
22     elif f(gjett) > 0:
23         print(f'f({gjett}) = {f(gjett)}, som er positivt.'
               )
24     elif f(gjett) < 0:
25         print(f'f({gjett}) = {f(gjett)}, som er negativ.')
26
27 else:  # Hvis løkka ikke avsluttes med break
28     print(f'Der har du prøvd {maksforsøk} ganger og gått
          tom for gjett!')
```

```
Jeg har tenkt på en funksjon som har et nullpunkt mellom 0
    og 100
Prøv å gjett det!


Gjett: 87
f(87.0) = 0.011204257850988442, som er positivt.

Gjett: 86
f(86.0) = -0.015646775666649293, som er negativ.

Gjett: 86.5
Helt riktig, f(86.5) er lik -0.0023846979469228 09!
Du brukte 4 forsøk!
```

3.2 Løse likninger numerisk

Vi fikk jo til å snu gjetteleken slik at datamaskinen spilte gjetteleken mot oss. Vi kan gjøre akkurat det samme med gjetteleken for å finne nullpunkt. Før vi snur dette på hodet må vi vite hvilke modifikasjoner vi skal gjøre.

Når vi vil finne nullpunkt til funksjonen så sjekker vi, som sagt, om $f(\texttt{gjett})$ er positiv eller negativ også oppdaterer vi grensene våre ute i fra hva svaret vårt er.

```
from pylab import log

def f(x):
    return (x/100)**5 - 0.1 * log((x + 1)/100) - 0.5

def midpunkt(a, b):
    return (a+b)/2

maks_tall = 100
min_tall = 0

maks_forsøk = 100
feilterskel = 0.01

for forsøk in range(1, maks_forsøk+1):
    gjett = midpunkt(min_tall, maks_tall)
    print(f'Jeg gjetter {gjett}!')

    if abs(f(gjett)) < feilterskel:
        print(f'Jippi, jeg greide det på {forsøk} forsøk,
           f({gjett})={f(gjett)}!')
        break
    elif f(gjett) > 0:
        maks_tall = gjett
    else:
        min_tall = gjett
else:
    print(f'Jeg brukte opp alle {maks_forsøk} forsøk uten
       å finne nullpunktet, det er et sted mellom {
       min_tall} og {maks_tall}')
```

```
Jeg gjetter 50.0!
Jeg gjetter 75.0!
Jeg gjetter 87.5!
Jeg gjetter 81.25!
Jeg gjetter 84.375!
Jeg gjetter 85.9375!
Jeg gjetter 86.71875!
```

```
Jippi, jeg greide det på 7 forsøk, f(86.71875)
   =0.0035199198002623 39!
```

Som vi ser fungerte dette veldig bra. Datamaskinen fant nullpunktet på 7 forsøk. Men, det er et problem med denne koden. For å illstrurere dette prøver vi å finne nullpunktet til funksjonen $g(x) = -f(x)$.

```
Jeg gjetter 50.0!
Jeg gjetter 25.0!
Jeg gjetter 12.5!
...
Jeg gjetter 3.155443620884047e-28!
Jeg gjetter 1.5777218104420236e-28!
Jeg gjetter 7.888609052210118e-29!
Jeg brukte opp alle 100 forsøk uten å finne nullpunktet
```

La oss undersøke hva som skjer her. $g(50) = 0.4$, og derfor blir øvre grense satt til å være femti. Dette er feil, nullpunktet er jo i $x \approx 86$! Koden burde ha flyttet den nedre grensa, ikke den øvre. Grunnen til dette er at vi naivt tror at funksjonen er positiv i $x = 100$ og negativ i $x = 0$. Dette stemmer ikke for vår funksjon g, og dermed virker ikke koden.

Hvordan kan vi fikse dette? Vi trenger en annen måte å bestemme om vi skal flytte øvre eller nedre grense? Dette kan vi gjøre ved å sjekke *fortegnet* til gjettet vårt. Hvis fortegnet til `f(gjett)` er likt fortegnet til `f(maks_tall)` betyr det at `gjett` og `maks_tall` er på samme side av nullpunktet. Altså må nullpunktet ligge mellom `min_tall` og `gjett`. Vi flytter derfor den øvre grensen. Dersom fortegnet til `f(gjett)` istedet er likt fortegnet til `f(maks_tall)`, flytter vi den nedre grensen.

For finne fortegn for et tall importerer vi `sign` fra pylab. Sign er en funksjon som tar inn et tall og returnerer 1 dersom det er et negativt tall og -1 dersom det er et positivt tall.

Her er den ferdige koden:

```
1 from pylab import sign, log
2
3 def f(x):
4     return (x/100)**5 - 0.1 * log((x + 1)/100) - 0.5
5
6 def g(x):
7     return -f(x)
```

```
8
9  def midpunkt(a, b):
10     return (a+b)/2
11
12 maks_tall = 100
13 min_tall = 0
14
15 maks_forsøk = 100
16 feilterskel = 0.01
17
18 for forsøk in range(1, maks_forsøk+1):
19     gjett = midpunkt(min_tall, maks_tall)
20     print(f'Jeg gjetter {gjett}!')
21
22     if abs(g(gjett)) < feilterskel:
23         print(f'Jippi, jeg greide det på {forsøk} forsøk,
               f({gjett})={f(gjett)}!')
24         break
25     elif sign(g(gjett)) == sign(g(maks_tall)):
26         maks_tall = gjett
27     else:
28         min_tall = gjett
29 else:
30     print(f'Jeg brukte opp alle {maks_forsøk} forsøk uten
          å finne nullpunktet')
```

```
Jeg gjetter 50.0!
Jeg gjetter 75.0!
Jeg gjetter 87.5!
Jeg gjetter 81.25!
Jeg gjetter 84.375!
Jeg gjetter 85.9375!
Jeg gjetter 86.71875!
Jippi, jeg greide det på 7 forsøk, f(86.71875)
   =0.0035199198002623 39!
```

Da har vi en fungerende likningsløser!

3.3 Visualisere midpunktmetoden med plotting

Her har vi en liten bonusseksjon for de som er interesserte i å visualisere midtpunktsmetoden. Det vi ønsker er å se en video av gjettene til algoritmen. For å få til det så må vi lage nesten samme plott for hvert gjett. Altså må vi legge til en plottekode inne i løkka. La oss gjøre det og forklare hva som skjer i koden etterpå.

```
from pylab import *

def f(x):
    return (x/100)**5 - 0.1 * log((x + 1)/100) - 0.5

def g(x):
    return -f(x)

def midtpunkt(a, b):
    return (a+b)/2

maks_tall = 100
min_tall = 0

maks_forsøk = 100
feilterskel = 0.01

x = arange(0, 100, 0.1)
y = g(x)

for forsøk in range(1, maks_forsøk+1):
    gjett = midtpunkt(min_tall, maks_tall)
    print(f'Jeg gjetter {gjett}!')

    cla()
    plot(x, y)
    plot(gjett, g(gjett), 'ro')
    title(f'Gjett nr: {forsøk}')
    pause(1)

    if abs(g(gjett)) < feilterskel:
        print(f'Jippi, jeg greide det på {forsøk} forsøk,
           f({gjett})={f(gjett)}!')
        break
```

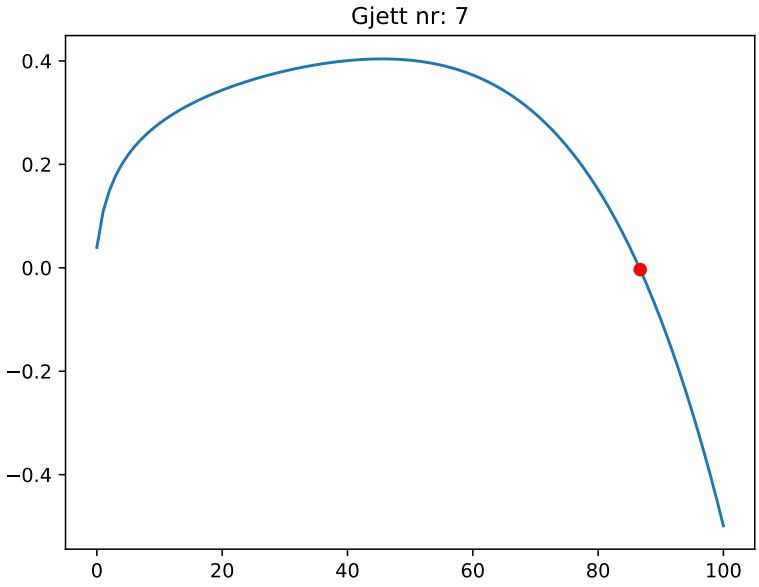
```
34      elif sign(g(gjett)) == sign(g(maks_tall)):
35          maks_tall = gjett
36      else:
37          min_tall = gjett
38  else:
39      print(f'Jeg brukte opp alle {maks_forsøk} forsøk uten
           å finne nullpunktet')
```

Forskjellen mellom denne plottekoden, og den vi hadde før, er at vi nå har regnet ut x og y verdier for hele intervallet vi leter etter nullpunktet i. Vi må gjøre dette for å kunne plotte kurva vår. Etter å ha gjort det, så kan vi se på den interessante nye biten av koden vår, nemlig linje 25-29, som vi også kan se her:

```
25      cla()
26      plot(x, y)
27      plot(gjett, g(gjett), 'ro')
28      title(f'Gjett nr: {forsøk}')
29      pause(1)
```

Noe her er kjent, vi plotter x mot y som et linjeplott i linje 26 av koden og vi plotter gjettet som en rød prikk i linje 27. Deretter setter vi tittelen til å være hvilket gjett-nummer vi er på.

Så har vi to linjer vi ikke har sett før. Linje 25, som har et kall til en `cla`-funksjon og linje 29, som har et kall til en `pause`-funksjon. Disse funksjonene er nyttige når vi lager plott som forandrer seg over tid. `cla` er en forkortelse for "clear axis", eller rengjør aksene. Denne funksjonen tilsvarer å ta et viskelær og viske vekk alt som er tegnet inne i akseobjektet vårt, slik at vi kan begynne å tegne på nytt. `pause` -funksjonen gjør en tilsvarende jobb som `show`-funksjonen — den viser frem det som er i plottevinduet. Men, istedenfor å bare vise det frem så passer `pause` også på å stoppe koden i en tidsperiode (`pause(1)` stopper i ett sekund og `pause(0.5)` stopper i et halvt sekund), slik at vi faktisk kan se på plottet før det endrer seg igjen. Under kan du se en figur som ble generert med plottekoden over:

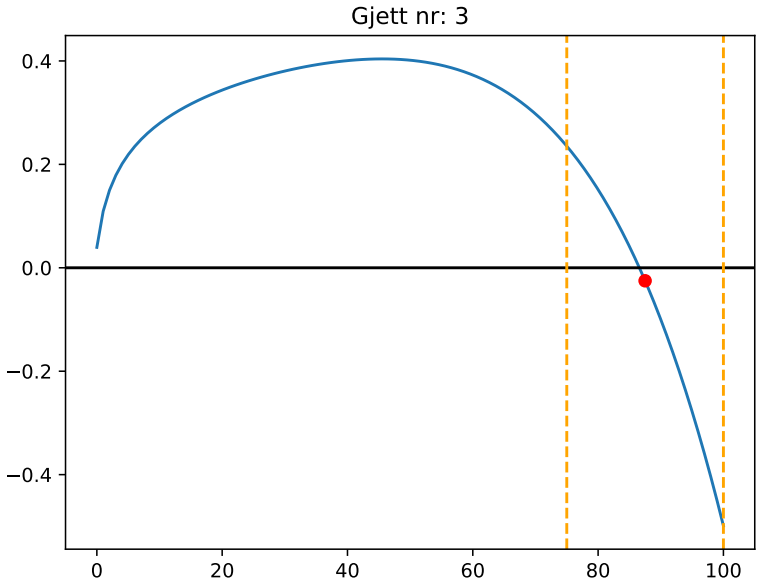


La oss nå se om vi kan gjøre plottet enda penere! Det første vi ønsker er å gjøre x-aksen tydeligere. For å gjøre det må vi trekke en rett horisontal strek i $y = 0$. Dette gjør vi med `axhline` funksjonen, som tar inn y-koordinaten og fargen vi ønsker at linja skal ha som argument. Så vil vi ha to vertikale stiplette linjer for å illustrere hvor søksområdet vårt er. For å få til det bruker vi den tilsvarende `axvline` funksjonen. Denne virker på samme måte so `axhline`, men tar inn x-koordinaten vi vil at den vertikale linja skal ligge på. Under har vi litt eksempelkode.

```
from pylab import *

def f(x):
    return (x/100)**5 - 0.1 * log((x + 1)/100) - 0.5

def g(x):
    return -f(x)

def midtpunkt(a, b):
    return (a+b)/2

maks_tall = 100
min_tall = 0

```

```
15 maks_forsøk = 100
16 feilterskel = 0.01
17
18 x = linspace(0, 100, 100)
19 y = g(x)
20
21 for forsøk in range(1, maks_forsøk+1):
22     gjett = midtpunkt(min_tall, maks_tall)
23     print(f'Jeg gjetter {gjett}!')
24
25     cla()
26     plot(x, y)
27     axhline(0, color='black')
28     axvline(min_tall, color='orange', linestyle='--')
29     axvline(maks_tall, color='orange', linestyle='--')
30     plot(gjett, g(gjett), 'ro')
31     title(f'Gjett nr: {forsøk}')
32     pause(1)
33
34     if abs(g(gjett)) < feilterskel:
35         print(f'Jippi, jeg greide det på {forsøk} forsøk,
              f({gjett})={f(gjett)}!')
36         break
37     elif sign(g(gjett)) == sign(g(maks_tall)):
38         maks_tall = gjett
39     else:
40         min_tall = gjett
41 else:
42     print(f'Jeg brukte opp alle {maks_forsøk} forsøk uten
          å finne nullpunktet')
```

Ved å kjøre dette programmet kan vi se hvordan algoritmen snevrer inn gjetteområdet rundt nullpunktet før den til slutt treffer målet. Prøv gjerne med forskjellige funksjoner og se hva som skjer. Under er en figur som viser hvordan figurene denne koden lager ser ut:


Gjett nr: 3
0.4
0.2
0.0
−0.2
−0.4
0
20
40
60
80
100