SymPy intro

Se også SymPy Intro, SymPy Numerisk, SymPy Programmering, SymPy Symbolsk,

Innledning

SymPy er Pythons beregningsverktøy for symbolsk matematikk. Dette heftet er et ambisiøst forsøk på å gå rett på sak og introdusere Sympy som et alternativ til å sjonglere med algebraiske uttrykk, løse likninger, derivere, integrere og løse andre problemer fra kalkulus og diskret matematikk.

Inntasting av uttrykk i SymPy gjøres på enlinjet form – og svarene vises også slik. Svarene kan til en viss grad omformes til uttrykk som er mer leselige med funksjonen pretty, du må likevel kunne lese uttrykk 'flytende' i SymPy hvis du skal arbeide effektivt – dessuten vil gjenbruk, klipp/lim av uttrykk bli enklere.

Du bør ha litt kjennskap til matematiske emner som algebra, likninger, kalkulus, rekker, laplacetransform, logikk for å få utbytte av kurset. Slå opp i din yndlingsmattebok hvis du trenger å friske opp kunnskapene.

I heftet er det vist eksempler kopiert inn fra fra arbeidsflaten i SymPy, noen ganger med små kommentarer som forklaring,
Python IDLE: P

Python IDLE etter oppstart:: 

```
from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)
>>> x = symbols('x')                    # Definerer x som symbolsk variabel
>>> integrate(x**3)                     # Beregner ∫x^3 dx
x**4/4
```

Prøv selv! Tast inn kommandoer i boksen nedenfor og kjør med Sympy Gamma.

integrate(x**3) SymPy

Eller med http://live.sympy.org/.

Du får best utbytte av kurset ved å ha startet opp SymPy og prøve ut kommandoene som blir vist i stedet for bare å lese teksten. Bruk klipp/lim fra kodeeksemplene. Lag dine egne varianter og se hva som skjer. Når mine knappe forklaringer ikke er tilstrekkelige vil du finne utfyllende hjelp bak noen tastetrykk i SymPy. I avsnittene som er merket *Gjør det selv* finner du treningsoppgaver som du kan prøve deg på.

Siden dette er en nettside kan du bruke søkefunksjonen Ctrl+F, F3 i nettleseren til å finne nøkkelord.

Gjør det selv 1

1. Skaff deg Python med SymPy og installer programvaren. Hvis dette er ditt første møte med Python kan det kanskje være greit å installere den komplette pakken Anaconda.
2. Et av utallig mange introkurs i Python er A Crash Course in Python for Scientists.
3. Et av nesten like mange introkurs til SymPy er Taming math and physics using SymPy.
4. Hvis du vil gå litt mer i dybden i materien kan du se på Introduction to scientific computing with Python.

Bakgrunnen til dette heftet er at SymPy brukes som beregningsmodul for pakken Symbolic i GNU Octave. Denne pakken er ment å være kompatibel med MATLABs syntaks for beregninger. Kommandoer i Octave blir der omgjort til SymPy syntaks - og resultatet av beregningen blir vist som i MATLAB. Denne kommunikasjonen mellom Octave og Sympy er ennå ikke 100% ferdig utviklet. De to beregningsverktøyene MATLAB Symbolic og SymPy kan utføre omtrent de samme operasjonene i matematiske emner som aritmetikk, algebra, kalkulus, rekker, mengder, logikk, men syntaksen i kommandoer er noe forskjellig. Inntil Octave Symbolic blir komplett kan SymPy være et alternativ.

Python og SymPy

Python IDLE etter oppstart, Sympy lastes inn:: 

```
import      laster inn moduler
math        modul for utvidet matematikk
cmath       utvidet matematikk med komplekse tall
```

Python er et generelt programmeringsspråk som kan utvides med moduler til å gjøre numeriske beregninger og algebraiske operasjoner. I utgangspunktet har det støtte for enkel aritmetikk, men moduler som `math`, `cmath` og `sympy` og andre gjør Python til et sterkt verktøy som takler symbolsk matematikk på høyt nivå. SymPy er altså standard Python utvidet med modul for symbolsk matematikk.

La oss teste Python på noen regnestykker

$1 + 2 \cdot 3 - \frac{4}{5} =$ $100^2 - 3^2 =$ $\frac{1}{\frac{2}{3}} - \frac{\frac{4}{5}}{6} =$

$1 + \frac{2 - \frac{3}{4}}{5 + \frac{6}{7}} =$ $8000000000 \times 2.5 \cdot 10^{-12} =$ $2^3 \cdot \sqrt{9} =$

```
>>> 1+2*3-4/5                          # Gangetegn er * og divisjonstegn er /
6.2
>>> 100**2-3**2                        # Eksponentoperator er **
9991
>>> 1/(2/3)-(4/5)/6
1.3666666666666667
>>> 1+(2-3/4)/(5+6/7)
1.2134146341463414
>>> 8e9 * 2.5e-12                      # Her brukes eksponentiell notasjon
0.02
>>> 2**3 * 9**(1/2)                    # Kvadratrot kan beregnes som potens
24.0
>>> abs(3+4j)                          # Komplekse tall har 1j som imaginær enhet
5.0
```

Kommandoer tastes inn etter promptet `>>>` og svaret vises på neste linje. Kommentarer starter med et hashtegn `#` og gjelder resten av linja. Komplekse tall bruker `1j` som imaginær enhet. Trigonometriske funksjoner er ikke støttet i utgangspunktet.

Evaluering skjer fra venstre mot høyre, men presedens overstyrer, operasjoner på høyere nivå utføres før operasjoner på lavere nivå. Uttrykk i parenteser beregnes først.

1.	( )	Parenteser
2.	**	Eksponenter
3.	+, -	Fortegn, pos, neg
4.	*, /, //, %	Multiplikasjon, divisjon, heltallsdivisjon, rest
5.	+ -	Addisjon og subtraksjon

```
>>> 8+5-6/2/3**-2+1                    # 13 - 3.0/(1/9) + 1 = -13.0
-13.0
>>> 8+(5-6/2)/3**-2+1                  # 8 + 2.0/(1/9) + 1 = 27.0
27.0
```

Python har mange moduler som utvider til avansert matematikk. Modulene `math` og `cmath` (for komplekse tall) lar oss utføre numeriske beregninger. Vi tar i bruk de utvidede funksjonene som ligger i en modul med kommandoen `import`. Vi kan videre enten bruke funksjonene i modulen ved å referere til disse enkeltvis, eller alle funksjonene under ett slik det vil bli gjort i dette kurset.

```
>>> from math import *                 # imporerer alt i modulen math
>>> 2**3 * sqrt(9)                     # 2^3√9
24.0
>>> pi
3.141592653589793
>>> sin(pi/6)                          # sinus til  π/6 radianer = 30 grader
0.49999999999999994                    # 'eksakt verdi'=1/2, π er avrundet.
>>> log(e**8)                          # Naturlig logaritme til e^8, burde bli 8,
7.999999999999999                      # men e er avrundet.
```

Python 3 skiller mellom divisjon med heltall og divisjon med reelle tall. Heltallsdivisjon gir heltall som resultat.

```
>>> 5/3                                  # operatoren / gir flyttall svar
1.6666666666666667
>>> 5//3                                 # operatoren // gir heltallsdivisjon
1
>>> 5%3                                  # resten etter 5//3 er 2
2
```

Komplekse tall tastes inn på kartesisk form med `j` som imaginær enhet. I biblioteket `cmath` ligger funksjoner for å håndtere operasjoner

```
>>> 3+4j                                 # imaginær enhet er j
(3+4j)
>>> abs(3+4j)                            # absoluttverdi, lengde
5.0
>>> from cmath import *
>>> phase(3+4j)                          # vinkel, argument i radianer
0.9272952180016122
>>> exp(-pi*1j)                          # burde ha blitt eksakt -1
(-1-1.2246467991473532e-16j)
```

De komplekse tallene ovenfor er av datatypen complex i standard Python. Senere skal vi definere symbolske uttrykk, og da blir komplekse tall vist på en annen form. Vi ser det samme med for eksempel brøkuttrykk, der `4/5` vises som `0.8` i standard Python, men som `4/5` i SymPy.

Algebra

Symbolske tall og variabler

```
symbols      lager symbolske variabler
S, sympify   lager symbolsk uttrykk
```

Tallregning med heltallsbrøker, rotuttrykk og potenser gjøres i SymPy etter at regnestykkene er omformet til symbolske uttrykk. Funksjonen `sympify`, eller med kortformen `S`, henter inn uttrykket som en string og gjør det om til et symbolsk objekt. Vi prøver med uttrykkene

$$\frac{1+\frac{2}{3}}{4+\frac{5}{6}} =, \qquad \sqrt{3}\cdot\sqrt{18} =, \qquad 2^{-4}\cdot 4^{5}\cdot 8^{-2} =$$

```
from sympy import *                    # importerer alt i modulen sympy
>>> S('(1+2/3) / (4+5/6)')
10/29
>>> S('sqrt(3) * sqrt(18)')
3*sqrt(6)
>>> S('2**(-4) * 4**5 * 8**(-2)')
1
```

Vi prøver også med uttrykk som innehholder konstantene e, π eller andre 'eksakte' oppstillinger,

$$\tan\tfrac{\pi}{3} =, \qquad \sin^{-1}\tfrac{1}{2} =, \qquad e^{\pi\cdot i} =$$

```
>>> S('tan(pi/3)')
sqrt(3)
>>> S('asin(1/2)')
pi/6
>>> S('exp(pi*I)')
-1
```

I algebra og calculus skal vi håndtere 'ukjente' størrelser og funksjoner av disse på generell form. Vi gir SymPy beskjed om det ved å bruke symbolske variabler som defineres med funksjonen symbols:

```
>>> from sympy import *
>>> x = symbols('x')                    # x er symbolsk variabel
>>> y, z, t = symbols('y z t')          # flere variabler kan defineres samtidig
>>> q = sqrt(y**2 + x**2)               # lager et symbolsk uttrykk
>>> p = q**2
>>> p
x**2 + y**2
```

Her er det laget 4 symbolske variabler, den første i en kommando og de tre neste i samme kommando. Variablene `p` og `q`, er rett og slett automatisk gitt status som symbolske variabler fordi de er tilordnet uttrykk der de symbolske variablene `x` og `y` inngår.

Når vi har med symbolske uttrykk å gjøre vil vi få svar med kvadratrøtter, heltallsbrøker, eksponenter, π, e, og andre formelelementer. Svarene på enlinje form kan da bli noe vanskelig å se for seg som 2D uttrykk. Til hjelp kan du bruke funksjonen pprint - som viser svaret med enkel bokstavgrafikk, eller funksjonen latex som gir LaTeX code til bruk i tekstbehandlere.

```
>>> S('acos(-1/sqrt(2))')
3*pi/4
>>> pprint(_)                           # underscore, _, er sist beregnede verdi
3·π
──
 4
>>> latex(_)
'\\frac{3 \\pi}{4}'
```

Når du blir litt dreven i enlinje syntaks blir du fort vant til å 'se' hvordan uttrykket er satt opp. Koden som latex gir ser slik ut i tekstbehandler: $\frac{3\,\pi}{4}$

Gjør det selv 2

Svar 2

SymPy kan også lage symbolske uttrykk ved å sette disse inn som tekststrenger, men det er ikke anbefalt. Likevel bør du kjenne til dette fordi det er fortsatt mye brukt og finnes i script og litteratur. Her er det symbolske uttrykket $q^2 - q - 2$ faktorisert,

```
>>> factor(S('q**2-q-2'))
(q - 2)*(q + 1)
```

- og svaret er et symbolsk uttrykk. Nedenfor gjøres det samme på den anbefalte måten, først defineres en symbolsk variabel `x` og deretter faktoriseres samme symbolske uttrykk som foran,

```
>>> x = symbols('x')
>>> factor(x**2-x-2)
(x - 2)*(x + 1)
```

Det greieste er å først definere de enkelte variablene med `symbols`, og så sette sammen uttrykket etterpå.

Symbolske variable representerer tall hentet fra tallmengden $\boldsymbol{C}$, komplekse tall. Noen ganger kan det bli aktuelt å anta bare relle tall, eller kanskje bare heltall som løsninger på problemene. Vi kan forutsette at en variabel bare skal være reell eller heltall eller positiv ved å tilføye `'real=True'` eller `'integer=True'` eller `'positive=True'` etter variabelnavnet i definisjonen. Med funksjonen assume kan vi også sette begrensinger i verdiområder. Hvis SymPy tilsynelatende ikke kan finne 'korrekt' løsning på et problem kan det være fordi vi ikke er oppmerksom på begrensninger.

Regneuttrykk

```
simplify          gjør forenklinger - om mulig
expand            utvider, multipliserer ut faktorer
factor            faktoriserer uttrykk
factor_list       lister opp faktorer
collect           samler ledd av samme grad for den ukjente
cancel            forkorter algebraiske brøker
apart             utfører delbrøkspalting
trigsimp          forenkler etter trigonometriske regler
expand_trig       utvider trigonometriske uttrykk
```

I algebra møter vi uttrykk på faktorisert form, eks $f(x) = (x - \frac{1}{2})(3 - 4x)(4 - 8x)$
eller på polynomform, $g(x) = 32x^3 - 56x^2 + 32x - 6$
eller som brøkuttrykk, $h(x) = \frac{3x-4}{x^2-x+6}$

SymPy kan forenkle, utvide, samle, forkorte, skrive om slike uttrykk ved hjelp av funksjonene ovenfor.

```
>>> f=(x-S('1/2'))*(3 - 4*x)*(4-8*x)   # Brøken 1/2 må gjøres om til symbolsk objekt
>>> expand(f)
32*x**3 - 56*x**2 + 32*x - 6
>>> factor(f1)
2*(2*x - 1)**2*(4*x - 3)
>>> simplify(f)
(2*x - 1)**2*(8*x - 6)
>>> h=(3*x-4)/(x**2 - x - 6)
>>> factor(h)
(3*x - 4)/((x - 3)*(x + 2))
>>> h1=apart(h)                        # h1 er et nytt symbolsk uttrykk
>>> h1                                 # navnet alene gir utskrift
2/(x + 2) + 1/(x - 3)
>>> simplify(h-h1)                     # sjekker om h og h1 er likeverdige uttrykk
0
```

Når du bruker funksjoner til å forenkle uttrykk er det ikke sikkert at Sympy gjør det på den måten du selv (eller fasit) vil mene er enkleste form.

Tall og uttrykk til variabler

```
= (likhetstegn)     tilordningsoperator
subs                substituerer variabler
```

En variabel er et navn som refererer til en verdi eller et symbolsk uttrykk. Tall, regneuttrykk og andre dataverdier tilordnes variabler med operatoren likhetstegn, =. Det gir den fordelen at verdier og uttrykk kan brukes på nytt uten å taste inn hele regnestykket. Lag navn på variablene som forteller hva de står for. Skal vi regne ut volumet av en rett sylinder kan vi for eksempel bruke r for radius, h for høyde og v for volum i formelen $v = \pi r^2 h$. Her beregnes volumet av en sylinder med r=2 og h=3:

```
>>> r = 2
>>> h = 3
>>> V = pi * r**2 * h
>>> V
12*pi                                  # Symbolsk uttrykk
>>> V.evalf()                          # Evaluerer til numerisk verdi
37.6991118430775
```

Likhetstegnet brukes altså som *tilordningsoperator* i SymPy i setninger som

variabelnavn = *verdi* eller *uttrykk*

der verdien eller resultatet av beregningen på høyre side blir lagret og gitt variabelnavnet på venstre side, slik at for eksempel en setning som `alder = alder + 1` virker snodig hvis det feilaktig leses som en matematisk likning, men som tilordning betyr det at verdien som variabelen `alder` refererer til økes med 1. En tilordning som `radius = 3` leses som 'radius tilordnes 3' eller 'radius får verdien 3'.

Variabelnavn må begynne med en bokstav og være uten mellomrom eller særnorske bokstaver. Med tegnet underscore kan vi lage oppdelte navn, for eksempel `svar_5_b` . SymPy skiller mellom store og små bokstaver. Resultatet av siste utførte beregning blir automatisk lagt i variabelen _ (underscore).

Et matematisk uttrykk som $\frac{3x-4}{x^2-x-6}$ kan også legges i en variabel,

```
>>> teller = 3*x-4
>>> nevner = x**2-x-6
>>> broek = teller/nevner
>>> solve(teller)                       # finner nullpunkter i teller
[4/3]
>>> solve(nevner)                       # finner nullpunkter i nevner
[-2, 3]
>>> factor(broek)                       # faktoriserer både teller og nevner
(3*x - 4)/((x - 3)*(x + 2))
>>> broek.subs(x, 2)                    # settter inn x=2 i broek
-1/2
>>> broek.subs(x, 3)                    # settter inn x=3 i broek,
zoo                                     # og får 'kompleks uendelig'
```

der broek er bygget opp av teller/nevner. Her ser vi også at det er mulig å substituere (erstatte) variabelen `x` med konkrete tall i uttrykket for broek, men syntaksen er kanskje littt spesiell. Først settes variabelnavnet, deretter et punktum, og så funksjonen `subs(x, 1)` som substituerer `x` med `1`.

Gjør det selv 3

Svar 3

Komplekse tall

Komplekse tall i SymPy kan tastes inn på normalform eller eksponentform, imaginær enhet er I,

```
>>> z = 1 + 2*I                         # I SymPy brukes I som imaginær enhet
>>> z
1 + 2*I
>>> w = sqrt(3)*exp(I*pi/2)             # Eksponentform, w = √3 e^(πi/2)
>>> w
sqrt(3)*I
>>> x = re(w)                           # reell komponent
>>> x
1
>>> y = im(w)                           # imaginær komponent
>>> y
2
```

Likninga $z^3 = -8$ har 3 komplekse tall som røtter. Tredjerota av -8, $\sqrt[3]{-8}$ med funksjonen `root(-8, 3)` vil SymPy viser som den rota som har (absolutt) minste faseverdi. De 3 tredjerøttene finner vi ved å løse likningen $z^3 = 8$,

```
>>> root(-8, 3)                         # Tredjerota av -8
2*(-1)**(1/3)
>>> N(_)
1.0 + 1.73205080756888*I
>>> z = symbols('z')
```

```
>>> solve(z**3+8)                       # løser likninga z^3 = -8
[-2, 1 - sqrt(3)*I, 1 + sqrt(3)*I]
```

Delbrøkspalting

```
apart          utfører delbrøkspalting
fraction       henter ut teller og nevner i en brøk
```

SymPy tar delbrøkspalting på strak arm. Her spaltes $\frac{7x+2}{x^3-3x^2+2x}$ til en sum av delbrøker:

```
>>> broek1 = (7*x + 2)/(x**3 - 3*x**2 + 2*x)
>>> pprint(broek1)
    7*x + 2
---------------
 3      2
x  - 3*x  + 2*x
>>> apart(broek1)
-9/(x - 1) + 8/(x - 2) + 1/x
>>> pprint(_)
    9       8     1
- ----- + ----- + -
  x - 1   x - 2   x
```

Vi kan hente ut teller og nevner i en brøk hver for seg med funksjonen `fraction`. Her er vi ute etter å finne nullpuktene i nevneren på `broek1`,

```
>>> fraction(broek1)
(7*x + 2, x**3 - 3*x**2 + 2*x)
>>> tn = fraction(broek1)
>>> tn[0]
7*x + 2
>>> tn[1]
x**3 - 3*x**2 + 2*x
>>> solve(tn[1])
[0, 1, 2]
```

Polynomdivisjon

```
div            utfører polynomdivisjon
```

En polynomdivisjon gir svaret i form av to uttrykk, kvotient og rest,

$$2x^3 + 3x - 4 \ : \ x - 1 \ = \ 2x^2 + 2x + 5 \ \ og \ \ rest = 1$$

eller skrevet om som brøkuttrykk,

$$\frac{2x^3+3x-4}{x-1} = 2x^2 + 2x + 5 + \frac{1}{x-1}$$

Hvis polynomdivisjonen 'går opp' betyr det at telleruttrykket har nevneruttrykket som faktor,

$$3x^3 - x^2 - 10x + 8 \ : \ x + 2 \ = \ 3x^2 - 7x + 4 \ \ og \ \ rest = 0$$

der en rest lik null gir at

$$3x^3 - x^2 - 10x + 8 \ = \ (3x^2 - 7x + 4)(x + 2)$$

Komplekse tall kan også være med i polynomene,

$$z^3 - (3 + i)z^2 - (4 - 3i)z + 4i \ : \ z - 1 \ = \ z^2 - 3z - 4 \ \ og \ \ rest = 0$$

I SymPy gir vi funksjonen `div` de to uttrykkene som skal divideres (teller og nevner) og resultatet i form av `kvotient` og `rest` henter vi til to variabler:

```
>>> kvotient, rest = div(2*x**3 + 3*x - 4, x - 1)
>>> kvotient
2*x**2 + 2*x + 5
>>> rest
1
>>> k, r = div(3*x**3 - x**2 - 10*x + 8, x + 2)
>>> k
3*x**2 - 7*x + 4
>>> r
0
>>> k, r = div(z**3 - (3 +I)*z**2 - (4 - 3*I)*z + 4*I, z-I)
>>> k
z**2 - 3*z - 4
>>> r
0
```

Her ser du eksempel på at SymPy gir to verdier som resultat av en funksjonsberegning, og vi må 'fange opp' begge verdiene til to variabler som vi selv dikter opp navn på, `k` og `r`. Slik kan vi vise at `k` og `r` inneholder 'antall hele ganger' og 'rest':

```
>>> teller = 2*x**3 + 3*x - 4; nevner = x - 1; k, r = div(teller, nevner);
>>> k*nevner+r
(x - 1)*(2*x**2 + 2*x + 5) + 1
>>> expand(_)
2*x**3 + 3*x - 4
```

Egne funksjoner

I mattetimene lærte du at en funksjon er en regel som knytter sammen verdier i en definisjonsmengde og en resultatmengde. En funksjon som finner kvadratrota av et tall kan vi kanskje kalle `rot`, og regelen kan være den matematiske operasjonen å opphøye tallet i 1/2. Uttrykket `rot(9)` henter tallet 9 fra definisjonsmengden og gir 3 til resultatmengden.

I SymPy kan vi lage symbolske uttrykk der vi kan sette inn (substituere) verdier for de symbolske variablene. Et uttrykk til å beregne kvadratrot kan settes opp slik:

```
>>> rota = x**(S('1/2'))                # 1/2 som symbolsk brøk
>>> rota.subs(x, 9)                     # Setter inn 9 for x
3
>>> rota.subs(x, 8)
2*sqrt(2)
```

Men, dette er altså et symbolsk uttrykk og ikke en funksjon. Funksjoner i Python lages med en `def`-setning der funksjonsnavn, inngangsverdier og beregningsregel og returverdi settes opp i et bestemt mønster. Her er en funksjon `rotb` som beregner kvadratrot:

```
>>> def rotb(x):                        # Funksjonsnavn og argument
        return x**(S('1/2'))            # Returnerer resultat av beregning
                                        # Tom linje avslutter
>>> rotb(9)
3
>>> rotb(8)
2*sqrt(2)
```

Funksjonen henter inn en x-verdi og *returnerer* (leverer fra seg) resultatet av beregningen. Dersom vi for eksempel vil lage en funksjon med med to variabler (`katet1`, `katet2`) som input må vi angi det,

```
>>> def hypotenus(katet1, katet2):
        return sqrt(katet1**2 + katet2**2)

>>> hypotenus(2, 4)
2*sqrt(5)
```

SymPy lar oss utføre operasjoner som derivasjon eller integrasjon med symbolske funksjoner. Vi må definere en symbolsk variabel som vi legger inn som funksjonsargument,

```
>>> x = symbols('x')
>>> def g(x):
        return 5*x**4-3*x**2

>>> diff(g(x), x)
20*x**3 - 6*x
>>> diff(g(x), x, 3)
120*x
>>> integrate(g(x))
x**5 - x**3
```

Selve formeluttrykket i en funksjon kan vises ved å taste funksjonsnavnet alene,

```
>>> g(x)                                # Funksjonsnavn med argument alene
5*x**4 - 3*x**2                         # viser funksjonsdefinisjonen
```

Likninger og ulikheter

Likninger

`solve`	generell likningsløser løser også ulikheter

Funksjonen solve løser likninger og ulikheter. Det brukes ikke likhetstegn mellom høyre og venstre side i en likning, likningen må ordnes til at en av sidene er null. Nedenfor er `solve` brukt til å løse symbolske likninger med en eller flere variable. Det er absolutt å anbefale at du slår opp hjelpesiden for `solve` og ser på detaljer for ulike typer likninger og løsninger.

Her løses likningene

$$2x + 1 = 3(x - 1)$$

$$x^3 - 2x^2 - x + 2 = 0$$

```
>>> x = symbols('x')
>>> solve(2*x+1-(3*(x-1)))              # Ikke VS=HS, men VS-HS=0
[4]                                     # Svaret leveres i en liste
>>> svar1 = solve(2*x+1-(3*(x-1)))      # Setter navn på svaret
>>> svar1[0]                            # og plukker ut det 0'te elementet
4
>>> likning2 = x**3-2*x**2-x+2          # Legger likningen i et symbolsk uttrykk
>>> svar2 = solve(likning2)
>>> svar2
[-1, 1, 2]
>>> likning2.subs(x, svar2[2])          # Tester om x=2 er løsning
0
```

Solve gir løsningen på likninger med *en* ukjent i en liste, og vi må plukke ut de enkelte elementene med indeksering (som starter med 0).

Her løses likningene

$2mh = 7(h - m)$ først med hensyn til h, deretter mht m

$t_F = \frac{9}{5}t_C + 32$ med hensyn til t_C.

```
>>> m, h, tC, tF = symbols('m h tC tF')
>>> svar3 = solve(2*m*h - 7*(m-h), h)
>>> svar3
[7*m/(2*m + 7)]
>>> svar4 = solve(2*m*h - 7*(m-h), m)
>>> svar4
[-7*h/(2*h - 7)]
>>> svar5 = solve(2*m*h - 7*(m-h))
>>> svar5
[{h: 7*m/(2*m + 7)}]
>>> tC, tF = symbols('tC tF')
>>> svar6 = solve(tF - S('9/5')*tC - 32, tC)
>>> svar6
[5*tF/9 - 160/9]
```

Røttene i likningene gis i en liste som i Python indekseres fra 0 og oppover. En dobbelrot vil gis som en verdi og en liking uten løsning gir en tom liste, [] som resultat.

Likninger som inneholder konstanter og bare en variabel blir løst med hensyn til denne variabelen. I likninger med to eller flere variable vil SymPy løse med hensyn til de variable som ligger alfabetisk først. I den tredje likninga er det h som automatisk velges som løsningsvariabel. Skal den samme likninga løses mht h må vi angi dette. I den siste likninga spesifiseres at den skal løses med hensyn til t_C, og der er også resultatet tilordnet t_C.
Det ville vel kanskje vært fristende å sette opp løsningen av den siste likninga som `tC=solve(tF-S('9/5')*tC-32,tC)`, men da gjør vi en tabbe. Vi bytter da type for variabelen tC fra symbol til liste.

Et likningssett med 3 likninger,

```
y + 2z = 4
x + z = 1
x + 2y + 4z = 2
```

løses slik:

```
>>> S = solve([y + 2*z - 4, x + z - 1, x + 2*y + 4*z - 2])
>>> S
{z: 7, x: -6, y: -10}
>>> S[x]
-6
```

Funksjonen solve tar de tre symbolske likningene inn som argumenter og svaret hentes i en vektor med tre elementer. Svaret leveres i en ordliste (dictionary). De enkelte løsningene hentes ut med den ukjente som nøkkel. Noen likningssett har flere løsninger, for eksempel

```
>>> svar = solve([y - 2*x, 4*x - y**2])
>>> svar
[{y: 0, x: 0}, {y: 2, x: 1}]
```

I dette tilfellet vil {x=0, y=0} være et løsningspar og {x=1, y=2} et annet løsningspar. I dette tilfellet har jeg hentet svaret til en datastruktur `svar` som vil bli en liste med to oppslagstabeller (dictionaries). Vi finner y-verdien i andre løsningspar ved å indeksere, først i svarlista, deretter i oppslagstabellen,

```
>>> svar[1][y]
2
```

Den ryddigste måten å arbeide med likninger er å legge de inn i en vektor som etterpå leveres til `solve`. Her viser jeg at verdiene i `svar` satt inn i likningene gir løsninger:

```
>>> likninger = [y - 2*x, 4*x - y**2]
>>> svar = solve(likninger)
>>> svar
[{y: 0, x: 0}, {y: 2, x: 1}]
>>> svar[1][x]                          # x-verdi for 2. løsningssett
1
>>> svar[1][y]                          # y-verdi for 2. løsningssett
2
>>> svar[1]
{x: 1, y: 2}
>>> likninger[1]
4*x - y**2
>>> likninger[0].subs(Svar[1])          # Sjekker om x=1 og y=2 er løsning
0                                       # på 1. likning
>>> likninger[1].subs(Svar[1])          # Sjekker om x=1 og y=2 er løsning
0                                       # på 2. likning
```

Ulikheter

Ulikheter bruker sammenligningsoperatorene >, <, >= og <= og løses med funksjonen solve. Nedenfor er det vist hvordan disse ulikhetene løses:

$3x - 5 > 2 + x$ $\qquad$ $3w + 2 \leq \frac{1}{w}$ $\qquad$ $|\frac{2q}{q+3}| < 1$

```
>>> solve(3*x-5 > 2+x, x)
x > 7/2
>>> w = symbols('w', real='True')
>>> solve(3*w+2 <= 1/w)
Or(And(-oo < w, w <= -1), And(0 < w, w <= 1/3))
#  w < -1   ELLER   0 < w <= 1/3
>>> q = symbols('q', real='True')
>>> a = solve(-1 < 2*q/(q+3))
>>> a
Or(And(-1 < q, q < oo), And(-oo < q, q < -3))
#  q > -1   ELLER   q < -3
>>> b = solve(2*q/(q+3) < 1)
>>> b
And(-3 < q, q < 3)
#  -3 < q < 3
# a og b har intervallet  -1 < q < 3  felles
```

De to siste ulikhetene forutsetter at `w` og `q` er i reelle tall. SymPy greide ikke å løse den siste ulikheten med absoluttverdifunksjonen. Derfor delte jeg den i to og satte sammen delløsningene til en felles løsning.

Eksempel 4.
En ball kastes rett opp og når maksimal høyde 6 m.
Hvor lenge var den over 2 m høyde?

Vi kan litt fysikk og bruker likningene

(1) $v = v_0 + at$

(2) $x = x_0 + v_0 t + \frac{1}{2}at^2$

Vi setter her x_0=0 og a=-g=-9.8 m/s^2. Her må vi vel gå noen runder med å løse likinger, først finner vi utgangshastigheten v_0 som gir en viss maksimal høyde, og deretter sette denne inn i likning 2 - for så å beregne tid når ballen passerer 2m høyde på opp- og nedtur.

```
>>> v, v0, x = symbols('v v0 x')
>>> t = symbols('t', positive=True)
>>> likning_1 = v - (v0 - 9.8*t)
>>> likning_2 = x - (v0*t - 9.8*t**2/2)
```

Utgangshastigheten og tiden til toppen kan finnes ved å løse disse to likningene ved å sette at x=6 og v=0. Vi lager et nytt likningssett med disse verdiene,

```
>>> likning_1a = likning_1.subs(v, 0)
>>> likning_2a = likning_2.subs(x, 6)
>>> v0t = solve([likning_1a, likning_2a], [v0, t])
>>> v0t
[(10.8443533693808, 1.10656667034498)]
```

Nå har vi utgangshastigheten som første element i en liste med tupler. Variabelen t er begrenset til positive verdier. Deretter lager vi en ny utgave av likning 2 der verdien av v_0 og høyde 2 m settes inn.

```
>>> v0n = v0t[0][0]
>>> likning_2b = likning_2.subs([(v0, v0n), (x, 2)])
>>> tider = solve(likning_2b, t)
>>> tid_over_2 = tider[1]-tider[0]
>>> tid_over_2
1.80701580581051
```

Calculus

```
limit           finner grenseverdi
diff            derivasjon
integrate       integrasjon, ubestemt og bestemt
```

Grenseverdi

Drøfting av funksjoner dreier seg noen ganger om problemstillinger som å finne hvilken verdi funksjonen går mot når den variable går mot en viss verdi. Et eksempel kan være å finne grenseverdien til uttrykket $\frac{\sin x}{x}$ når x går mot verdien 0. Definisjonsområdet er alle mulige reelle tall for x, bortsett fra x=0 fordi nevneren ikke kan ha verdien null. Funksjonen `limit` gjør jobben,

```
>>> limit(sin(x)/x, x, 0)
ans =
1
```

Vel og bra, men om vi forandrer litt på funksjonen, til $\frac{\sin x}{|x|}$, hva får vi da?

```
>>> limit(sin(x)/abs(x), x, 0, dir='+')
1
```

Kanskje det har noe med hvilken side av null x starter fra? Vi prøver fra venstre og så fra høyre:

```
>>> limit(sin(x)/abs(x), x, 0, dir='-')
ans =
-1
```

Det skulle stemme med teoriene, det behøver ikke være samme grenseverdi når x går mot testverdien fra høyre som når x går mot testverdien fra venstre.

Her er en rekkeutvikling av Eulers konstant, grunntallet for naturlige logaritmer, $e = \lim_{n\to\infty}(1+\frac{1}{n})^n$:

```
>>> n=7777; (1+1/n)**n                  # numerisk Python
2.718107084888141
>>> limit((1+1/n)**n, n, oo)
2.71810708488814
>>> m = symbols('m')
>>> limit((1+1/m)**m, m, oo)            # Symbolsk
E
```

SymPy bruker variabelen `oo` som 'verdi' på uendelig - og gir her svaret `E` som er likeverdig med `exp(1)`, $e^1 = e$.

Derivasjon

Derivasjon gjøres med funksjonen `diff`,

```
>>> diff(sin(x**2))
2*x*cos(x**2)
>>> diff(x**2 + 2*x*y - y**2, y)
2*x - 2*y
```

Det første differensialet utfører $\frac{d(\sin x^2)}{dx}$ og fordi x er eneste uavhengige variabel er det ikke nødvendig å angi det. Det andre differensialet utføres med hensyn til `y` og `x` betraktes som konstant.

Andre eller høyere ordens deriverte har ordenstallet med i funksjonsuttrykket,

```
>>> diff(sin(x), x, 4)
sin(x)
```

Integrasjon

Symbolsk integrasjon bruker funksjonen `integrate` som kan finne både ubestemt og bestemt integral, eks $\int \sin^2 x\, dx$, $\int_{-\pi}^{\pi} \sin^2 x\, dx$ og $\int x^n\, dx$,

```
>>> integrate((sin(x))**2)
x/2 - sin(x)*cos(x)/2
>>> integrate((sin(x))**2, [x, -pi, pi])
pi
>>> integrate(x**n, x)
Piecewise((log(x), n == -1), (x**(n + 1)/(n + 1), True))
```

Legg merke til at SymPy ikke setter inn en integrasjonskonstant i svarene. Det tredje integralet ovenfor er lik $\ln x$ for n=-1 og $\frac{x^{n+1}}{n+1}$ for n<>-1. Funksjonen som finner naturlig logaritme er `log(x)`.

I mange tilfeller arbeider vi med et uttrykk som er definert som en funksjon av en variabel, for eksempel $f(x) = \sin(3x+4)$, og integralet $\int f(x)\, dx$ utføres da som

```
>>> def f (x): return 2*sin(3*x+4)

>>> integrate(f(x))
-2*cos(3*x + 4)/3
>>> integrate(f(x), [x, -pi, pi])
0
```

Eksempel 5
Gitt funksjonen $f(x) = x^3 - 6x^2 + 9x$

a) Lag grafen til funksjonen fra x=-1 til x=5
b) Finn den deriverte av funksjonen
c) Finn topp- og bunnpunkt
d) Finn vendepunkt
e) Beregn arealet under grafen fra x=0 til x=3

a) Vi gir SymPy beskjed om å bruke symbolske variabler som defineres med funksjonen symbols. Den matematiske funksjonen $f(x)$ som vi skal behandle legger vi inn i et symbolsk uttrykk y. Et slikt uttrykk er ikke en funksjon i matematisk forstand, vi kan ikke beregne funksjonsverdien for x=1 som $f(1)$ eller $y(1)$, men ved å substituere (bytte ut) x med 2 i uttrykket `y.subs(x, 1)`,

```
>>> x = symbols('x')                    % Lager en symbolsk variabel x
>>> y = x**3-6*x**2+9*x                 % Lager et symbolsk uttrykk med
funksjonsdefinisjonen
```

Funksjonen plot tar saken, her for x-verdier i [-1, 5],

```
>>> plot(y, (x, -1, 5))                 % GJØR DET SELV!
```

b) Den deriverte finnes med funksjonen `diff`. Setter navnet `yD` på uttrykket for den deriverte,

```
>>> yD = diff(y)                        % fD er den deriverte av f(x)
>>> yD
3*x**2 - 12*x + 9
```

Vi har nå uttrykket for den deriverte av $f(x)$ og kan beregne stigningstallet for ulike x-verdier ved substitusjon.

c) Toppunktet kan ikke finnes med en enkelt kommando, men vi kan drøfte funksjonen på vanlig måte - først finne nullpunkter for den deriverte, og så avgjøre om de gir min- eller maks-verdier:

```
>>> np_yD = solve(yD)                   % Nullpunkter for den deriverte, bruker
symbolsk likningsløser
np_yD                                   % Nullpunktene ligger i vektoren np_yD
[1, 3]
>> yD2 = diff(y, x, 2)                  % Den andrederiverte av f(x) kalles yD2
>>> yD2
6*(x - 2)
```

Vi ser nå at den deriverte av $f(x)$ har to nullpunkter som er lagt inn i vektoren `np_yD`. Videre har vi funnet den andrederiverte som uttrykket `yD2`. Substituerer så x-verdiene for nullpunktene i den deriverte av `f(x)` inn i `yD2` for å finne fortegnet til krumningene,

```
>>> krum1 = yD2.subs(x, np_yD[0])       % Den andrederivertes verdi for 1.
nullpunkt i den deriverte
>>> krum1
-6
>>> krum2 = yD2.subs(x, np_yD[1])       % Den andrederivertes verdi for 2.
nullpunkt i den deriverte
>>> krum2
6
```

Nå begynner det kanskje å bli en smule kryptisk. Den siste kommandoen setter altså de to x-verdiene 1 og 3 inn for `x` i den andrederiverte og viser at x=1 gir et toppunkt (fordi krumningen er negativ), og tilsvarende et bunnpunkt for x=3. De to `x`-verdiene substitueres også inn i `f(x)` slik at vi finner koordinatene for topp- og bunnpunkt:

```
>>> x_topp = np_yD[0]
>>> x_bunn = np_yD[1]
>>> y_topp = y.subs(x, np_yD[0])
>>> y_bunn = y.subs(x, np_yD[1])
>>> ekstrema = (x_topp, y_topp), (x_bunn, y_bunn)
>>> ekstrema
((1, 4), (3, 0))
```

d) Vendepunktet er der den andrederiverte (grafens krumning) skifter fortegn. Velger her å finne dette ved å løse ulikheten yD2 > 0 ,

```
>>> x_vp = solve(yD2 > 0)
>>> x_vp
And(2 < re(x), im(x) == 0, re(x) < oo)
```

Intervallet der krumningen er positiv er x > 2. Negativ krumning for x < 2 og vendepunkt for x = 2. Koordinatene beregnes ved substitusjon,

```
>>> x_vp = solve(yD2)
>>> y_vp = y.subs(x, x_vp[0])
>>> vp = (x_vp[0], y_vp)
>>> vp
(2, 2)
```

e) Det bestemte integralet fra x=0 til x=3 finnes med funksjonen `int`,

```
>>> yIntegral = integrate(y, [x, 0, 3])
>>> yIntegral
27/4
```

Differensiallikninger

```
dsolve      løser differensiallikninger
```

Inntasting av differensiallikninger følger en syntaks som du skulle kunne kjenne igjen fra vanlige uttrykk. Difflikninga y' = y·x er en kortform for $\frac{df}{dx}(x) = f(x) \cdot x$ og løses slik:

```
>>> x = symbols('x')
>>> f = symbols('f', cls=Function)
>>> svar = dsolve(diff(f(x), x)-x*f(x), f(x))
>>> svar
Eq(f(x), C1*exp(x**2/2))
```

Her svarer SymPy med en likning (funksjonsuttrykk) på formen `Eq(VS, HS)` som vi i mattetimene ville ha skrevet som VS = HS, altså $f(x) = C \cdot e^{\frac{x^2}{2}}$. De to sidene i `svar` kan hentes ut som `svar.lhs` og `svar.rhs`. La oss teste om vi har funnet en løsning. Vi sjekker om venstre og høyre side i likningen er like.

```
>>> y = svar.rhs
>>> venstre = diff(y, x)
>>> venstre
C1*x*exp(x**2/2)
>>> hoyre = y*x
>>> hoyre
C1*x*exp(x**2/2)
```

Svaret inneholder den ubestemt konstanten `C1` (som i lærebøker heter bare C), men SymPy har antakelig sin egen interne ordning på disse. Den tilsvarende difflikninga med gitt startverdi y(0)=5 løses slik:

```
# Ikke mulig foreløpig?
```

Laplacetransform

```
laplace_transform             Finner laplace transform
inverse_laplace_transform     Finner invers laplace transform
```

En Laplace transform overfører en funksjon $f(t)$ til en funksjon $F(s)$ etter definisjonen
$F(s) = \int_0^\infty f(t) \cdot e^{-s \cdot t}\, dt$

Vi tester med funksjonene $f(t) = 1, \quad g(t) = t, \quad h(t) = e^{-t}, \quad k(t) = e^t \cdot \cos(2t)$:

```
>>> s = symbols('s')
>>> t = symbols('t', positive=True)
>>> f=1; g=t; h=exp(-t); k=exp(t)*cos(2*t)
>>> laplace_transform(f, t, s)
(1/s, -oo, 0 < re(s))
>>> laplace_transform(f, t, s, noconds=True)
1/s
>>> laplace_transform(g, t, s, noconds=True)
>>> laplace_transform(h, t, s, noconds=True)
1/(s + 1)
s**(-2)
>>> laplace_transform(k, t, s, noconds=True)
(s - 1)/((s - 1)**2 + 4)
```

De siste opplysningene i svarene er konvergensbetingelser som kan tas bort ved å tilføye noconds=True i kommandoen.

En enhetssteg funksjon, `Heaviside(t-a)` har verdien 0 fra t=0 til t=a - og verdien 1 for t>=a. En firkantpuls med verdi 7 fra t=3 til t=5 kan vi lage ved å sette sammen to enhetssteg funksjoner,

```
>>> laplace_transform(Heaviside(t-0), t, s, noconds=True)
1/s
>>> laplace_transform(7*(Heaviside(t-3)-Heaviside(t-5)), t, s, noconds=True)
(7*exp(2*s) - 7)*exp(-5*s)/s
```

Invers Laplace bruker funksjonen `inverse_laplace_transform`,

```
>>> inverse_laplace_transform((s - 1)/((s - 1)**2 + 4), s, t)
exp(t)*cos(2*t)
>>> inverse_laplace_transform((7*exp(2*s) - 7)*exp(-5*s)/s, s, t)
-7*Heaviside(t - 5) + 7*Heaviside(t - 3)
```

Her er et eksempel som tester om $L(L^{-1}(F(s))) = F(s)$, altså at om vi først gjør invers laplace transform på en s-funksjon, og deretter laplace transform på resultatet, så bør vi ende opp med den opprinnelige laplacetransformerte,

```
>>> F = (4*s**2 - 8*s + 2)/(s**3 - 3*s**2 + 2*s)
>>> f = inverse_laplace_transform(F, s, t)
>>> f
exp(2*t) + 2*exp(t) + 1
>>> G = laplace_transform(f, t, s, noconds=True)
>>> G
2*(2*s**2 - 4*s + 1)/(s*(s**2 - 3*s + 2))
>>> simplify(F-G)
0
```

Jo, det ser ut til å stemme, men legg merke til at SymPy kan presentere svar på alternative måter enn i fasit og lærebøker.

Plotting

```
matplotlib      avansert modul for plotting
plot            funksjon som lager plot
```

Python bruker modulen `matplotlib` til å lage grafer og figurer i mange formater. Grafer med akser, rutenett, forklaringer og andre elementer settes mest effektivt opp med kommandoer til Python. Matplotlib må installeres separat - eller følger med hvis du bruker en av de proffe distribusjonene som Anaconda, Enthought Canopy, Spyder. Modulen bruker matplotlib til å tegne grafer med kommandoer som er komatible med MATLAB. En kort introduksjon til plotting finner du i Matplotlib tutorial av Nicolas P. Rougier

Sympy bruker matplotlib til plotting med kommandoen plot. Hvis matplotlib ikke er installert vil du få se en helt enkel tekstbasert figur som antyder med prikker og streker hvordan grafene ser ut.

```
>>> x = symbols('x')
>>> plot(sin(x))
0.99935 |
        |            ...               ...               . .
        |           .   .             .   .             .   .
        |  .                         .     .           .     .
        |          .     .                                     
      0 | -.---------------------------.--------.---------.--------.--
        |   .     .        .        .         .         .       .
        |    .   .          .      .           .       .
        |     . .            .    .             .    .
-0.9993 |      ...            ...                ...
         -10                   0                          10
<sympy.plotting.plot.Plot object at 0x03ACD5B0>
```

Rekker

```
range              lager ordnet tallområde
for . in range     gjennomløper tallområde
sum                summerer ledd i liste
summation          summerer med formel for generelt ledd
limit              finner grenseverdi for generelt ledd
```

Tallfølger

En tallfølge er en endelig eller uendelig ordnet liste med tall. En posisjonsindeks n angir tallenes plass i følgen og verdien av det n-te leddet a_n er gitt av en regel (formel) som en funksjon av n, $a_n = a(n)$.

En endelig tallfølge skriver vi vanligvis på papiret innenfor krøllparenteser, {1, 1/2, 1/3, 1/4} - og en uendelig tallfølge kan angis med bare det generelle leddet, $\{a(n)\}$ eller {1/n}. I Python bruker vi datatypen list til å håndtere tallfølger - en kommaseparert liste innenfor hakeparenteser.

```
>>> [1, 1/2, 1/3, 1/4]                # datatypen list
[1, 0.5, 0.3333333333333333, 0.25]
>>> S('[1, 1/2, 1/3, 1/4]')           # symbolsk objekt av datatypen list
[1, 1/2, 1/3, 1/4]
>>> n = symbols('n', integer=True)
>>> a_n = 1/n                         #generelt ledd i den harmoniske tallfølgen, [1,
1/2, 1/3, 1/4, 1/5, 1/6,...]
>>> b_n = 1/n!                     # geometriske følge med k=1/2, [1/1, 1/2, 1/6,
1/24,1/120, 1/720,...]
>>> c_n = 1/factorial(n)              # nevnere er fakultetstallene
```

Det enkelte ledd som funksjon av indeksen n kan finnes ved substitusjon,

```
>>> a_n.subs(n: 4)
1/4
```

```
>>> c_n.subs(n: 4)
1/24
```

En list er en datatype i Python som lagrer dataverdier på en ordnet måte, altså egnet til å lagre en tallfølge. Lista kan vi enten sette opp ved oppramsing, `[0, 1, 2, 3, 4]` eller med funksjonen `range`,

```
#>>> range(5)                          # NB! Python 2.x: range lager en liste
#[0, 1, 2, 3, 4]                       # NB! Python 2.x
>>> range(5)
range(0, 5)                            # range kan lage enkeltverdier i liste
>>> list(range(5))                     # range fyller ut verdier i liste
[0, 1, 2, 3, 4]
>>> list(range(1, 9, 2))               # fra 1 til 9 i sprang på 2
[1, 3, 5, 7]
```

Funksjonen range er hendig å bruke til å lage tallfølgen 1,2,3,...,n som indekserer som en tallfølge. Ved å kombinere funksjonen range og en for-løkke får vi comprehension, som kan lage ledd i tallfølger med indeksverdiene fra range,

```
>>> [m for m in range(1, 5)]
[1, 2, 3, 4]
>>> [(1/n).subs(n, m) for m in range(1, 5)]
[1, 1/2, 1/3, 1/4]
```

Denne listeoperasjonen kan vi så bruke til å sette inn indeksene i uttrykket for leddene i a_n, b_n og c_n,

```
>>> [a_n.subs(n: i) for i in range(0,8)]
[zoo, 1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7]
>>> [b_n.subs(n: i) for i in range(0,8)]
[1, 1/2, 1/4, 1/8, 1/16, 1/32, 1/64, 1/128]
>>> [c_n.subs(n: i) for i in range(0,8)]
[1, 1, 1/2, 1/6, 1/24, 1/120, 1/720, 1/5040]
```

der i-verdiene 0-7 settes inn fortløpende for n. Vi ser at $a_0=1/0$ ikke er definert.

Tallfølger kan ha ledd som øker over alle grenser for store verdier av n, det er divergente tallfølger. Andre har ledd som nærmer seg en viss verdi for store verdier av n - det er konvergente tallfølger. De tre tallfølgene $\{a_n\}$, $\{b_n\}$ og $\{c_n\}$ ovenfor vil tydeligvis ha ledd som konvergerer mot 0. Andre tallfølger har ledd der det ikke er så gjennomskuelig å avgjøre divergens eller konvergens,

```
>>> d_n=(n-2)/(n-1)
>>> [d_n.subs(n: i) for i in range(2,10)]
[0, 1/2, 2/3, 3/4, 4/5, 5/6, 6/7, 7/8]
>>> f_n=n**3/2**n
>>> [f_n.subs(n: i) for i in range(0,8)]
[0, 1/2, 2, 27/8, 4, 125/32, 27/8, 343/128]
>>> g_n=(1-1/n)**(-n)
>>> [g_n.subs(n: i) for i in range(2,6)]
[4, 27/8, 256/81, 3125/1024, 46656/15625]
>>> limit(d_n, n, oo)
1
>>> limit(f_n, n, oo)
0
>>> limit(g_n, n, oo)
E
```

Rekker

En rekke er summen av leddene i en tallfølge. Vi finner summen av en endelig rekke med funksjonen `sum`,

```
>>> 1 + 1/2 + 1/3 + 1/4
2.083333333333333
>>> S('1 + 1/2 + 1/3 + 1/4')          # lager et symbolsk sumuttrykk
25/12
>>> s1=S('[1, 1/2, 1/3, 1/4]')        # lager en symbolsk tallfølge i en liste
>>> sum(s1)                           # som summeres
25/12
```

Rekker med formel for det generelle leddet summeres med funksjonen `summation`.
Her utføres summene $\sum_{n=1}^{5} \frac{1}{n}$, $\sum_{n=1}^{\infty} \frac{1}{n}$, $\sum_{n=1}^{5} \frac{1}{2^n}$ og $\sum_{n=1}^{\infty} \frac{1}{2^n}$.

```
>>> n = symbols('n', integer=True)
>>> a_n = 1/n
>>> summation(a_n, [n, 1, 5])
137/60
>>> summation(a_n, [n, 1, oo])
oo
>>> b_n = 1/2**n
>>> summation(b_n, [n, 1, 5])
31/32
>>> summation(b_n, [n, 1, oo])
1
```

Som vi ser ovenfor vil den harmoniske rekka 1/n få en sum som vokser over alle grenser for økende verdier av n, rekka divergerer. Den andre rekka, 'halveringsrekka' $1/2^n$, vil konvergere mot 1 i sum.

I forrige kapittel så vi at en tallfølge kan ha ledd som konvergerer mot en viss verdi. En uendelig rekke er en sum med uendelig mange ledd og denne kan også konvergere, for eksempel $\sum_{n=1}^{\infty} \frac{1}{n!}$:

```
>>> c_n = 1/factorial(n)
>>> summation(c_n, [n, 0, 12])
260412269/95800320
>>> summation(c_n, [n, 0, 12]).n()
2.71828182828617
>>> summation(c_n, [n, 0, oo])
E
```

Konvergens av rekker

Forholdstesten kan brukes til å finne ut om rekka $\sum_{n=1}^{\infty} \frac{n^2}{3^n}$ konvergerer, det vil si går mot en endelig sum når n vokser over alle grenser.

Forholdstresten går ut på å undersøke grenseverdien for forholdet mellom to ledd etter hverandre, $\lim_{n\to\infty} \frac{a_{n+1}}{a_n}$

```
>>> n = symbols('n', integer=True)
>>> a = n**2/3**n                     # Leddet a_n
>>> a1 = a.subs(n, n+1)               # Leddet a_n+1
>>> r = a1/a
>>> limit(r, n, oo)
1                                     # Bug i SymPy, burde bli 1/3 !!
>>> limit(expand(a1/a), n, oo)        # workaround
1/3
```

Forholdstesten viser at rekka ovenfor konvergerer, men hva blir summen?

```
>>> summation(a, [n, 1, oo])
3/2
```

Et eksempel til, vi skal finne konvergensintervallet til rekka med generelt ledd $a_n = \frac{(x-1)^n}{2^n(n+1)}$, og bruker forholdstesten med drøfting av $L = \frac{a_{n+1}}{a_n}$, der L er en funksjon av x,

```
>>> x = symbols('x')
>>> n = symbols('n', integer=True)
>>> a = (x-1)**n/(2**n*(n+1))
>>> a1 = a.subs(n, n+1)
>>> L = limit(a1/a, n, oo)
>>> L
1                                       # Bug i SymPy!!
>>> L = limit(apart(a1/a), n, oo)       # workaround
>>> L
x/2 - 1/2
```

Teorien om konvergens sier at vi skal drøfte uttrykket |L| < 1 og til og med teste endepunkter for å finne konvergensintervallet,

```
>>> solve(-1 < L , x)
x > -1.0
>>> solve(L < 1, x)
x < 3.0
```

Kombinert løsning av de to ulikhetene gir -1 < x < 3, og i intervallets nedre grensepunkt har vi rekka med ledd

```
>>> a.subs(x, -1)
(-2)**n*2**(-n)/(n + 1)
>>> simplify(_)
(-1)**n/(n + 1)
```

som er en alternerende harmonisk rekke – og den konvergerer, og så testes for x=3:

```
>>> b.subs(x, 3)
1/(n + 1)
```

som er leddene i en harmonisk rekke – og den divergerer. Konvergensintervall, $-1 \leq x < 3$.

Differenslikninger, rekursjonslikninger

```
  rsolve      Løser rekursjonslikninger
```

En differenslikning eller rekursjonslikning er en oppstilling som setter opp en sammenheng mellom leddene i en tallfølge. Her er utsnitt av en tallfølge, ... 7, 15, 31, 63, 127, 255, 511, ...

Sammenhengen mellom leddene er at *et ledd i sekvensen er lik 1 mer enn to ganger det foregående*, $a_{n+1} = 2 \cdot a_n + 1$, men det samme kan også sies om tallfølgen ... 5, 11, 23, 47, 95, 191, 383, ...

Å løse dette som en rekursjonslikning betyr å finne en generell formel for det n'te leddet i sekvensen, men som vi ser er den eksakte løsningen bare mulig om vi kjenner en startbetingelse som for eksempel at a_1=1 - ellers må vi nøye oss med en løsing basert på en ukjent konstant som startverdi.

Her er løsning på differenslikningen ovenfor, først uten startverdi, og så med med a_1=1:

```
>>> n = symbols('n', integer=True)
>>> a = symbols('a', cls=Function)
```

```
>>> rsolve(a(n+1)-2*a(n)-1, a(n))
2**n*C0 - 1
>>> rsolve(a(n+1)-2*a(n)-1, a(n), {a(1):1})
2**n - 1
```

Fibonaccisekvensen er basert på rekursjonslikningen $a_{n+2} = a_{n+1} + a_n, \quad a_1 = a_2 = 1$, og har den nesten ubegripelige løsningen

```
>>> Fn = rsolve(a(n+2)-a(n+1)-a(n), a(n), {a(1):1, a(2):1})
>>> Fn
sqrt(5)*(1/2 + sqrt(5)/2)**n/5 - sqrt(5)*(-sqrt(5)/2 + 1/2)**n/5
```

- altså tallfølgen $\frac{\sqrt{5}}{5}\left(\frac{1}{2}+\frac{\sqrt{5}}{2}\right)^n - \frac{\sqrt{5}}{5}\left(-\frac{\sqrt{5}}{2}+\frac{1}{2}\right)^n$ - et uttrykk med potenser av kvadratrota til 5 som alt i alt blir beregnet til heltall! La oss teste:

```
>>> >>> F7 = [Fn.subs(n, i) for i in range(1,8)]
>>> F7
[-sqrt(5)*(-sqrt(5)/2 + 1/2)/5    + sqrt(5)*(1/2 + sqrt(5)/2)/5,
 -sqrt(5)*(-sqrt(5)/2 + 1/2)**2/5 + sqrt(5)*(1/2 + sqrt(5)/2)**2/5,
 -sqrt(5)*(-sqrt(5)/2 + 1/2)**3/5 + sqrt(5)*(1/2 + sqrt(5)/2)**3/5,
 -sqrt(5)*(-sqrt(5)/2 + 1/2)**4/5 + sqrt(5)*(1/2 + sqrt(5)/2)**4/5,
 -sqrt(5)*(-sqrt(5)/2 + 1/2)**5/5 + sqrt(5)*(1/2 + sqrt(5)/2)**5/5,
 -sqrt(5)*(-sqrt(5)/2 + 1/2)**6/5 + sqrt(5)*(1/2 + sqrt(5)/2)**6/5,
 -sqrt(5)*(-sqrt(5)/2 + 1/2)**7/5 + sqrt(5)*(1/2 + sqrt(5)/2)**7/5]
>>> F7b = [simplify(Fn.subs(n, i)) for i in range(1,8)]
>>> F7b
[1, 1, 2, 3, 5, 8, 13]
```

Taylor/Maclaurin polynom

```
  series         Finner Taylorpolynom for funksjon
```

Det har vært kjent i 300 år at en funksjon som er uendelig mange ganger deriverbar kan representeres med en rekke.

Eksempel: $f(x) = \sqrt{1+x}$ som polynom rundt x=0 :

```
>>> series(sqrt(1+x))
1 + x/2 - x**2/8 + x**3/16 - 5*x**4/128 + 7*x**5/256 + O(x**6)
```

Det siste leddet, O(x**6) er et restledd og representerer avviket mellom de viste leddene og funksjonsverdien. Restleddet kan tas bort med metoden `removeO`. Hvis funksjonen `series` blir gitt et funksjonsuttrykk alene blir leddene opp til 5. orden beregnet, eller vi kan bestille flere ledd med et tilleggsargument,

```
>>> series(sin(x))
x - x**3/6 + x**5/120 + O(x**6)
>>> series(sin(x), x, 0, 10)
x - x**3/6 + x**5/120 - x**7/5040 + x**9/362880 + O(x**10)
>>> series(sin(x), x, 0, 10).removeO()
x**9/362880 - x**7/5040 + x**5/120 - x**3/6 + x
>>> _.subs(x, pi/2).evalf()
1.00000354258429                      # ≈ sin(π/2)
```

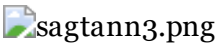
I siste kommando er x substituert med pi/2 og deretter vist som numerisk verdi med metoden `evalf`. Funksjonen `series` brukes også til rekkeutvikling rundt andre verdier enn x=0,

```
>>> series(1/x, x, 1, 5)
2 + (x - 1)**2 - (x - 1)**3 + (x - 1)**4 - x + O((x - 1)**5, (x, 1))
```

- som er Taylorpolynomet for 1/x rundt x=1 opp til orden 5. Legg merke til at SymPy er litt uryddig i presentasjonen av rekka. I lærebøker vil du finne det skrevet slik: `1 - (x-1) + (x-1)**2 - (x-1)**3 + ... .`

Fourierrekker

Fourierrekker brukes til å beskrive en periodisk funksjon som rekkeutvikling av cosinus- og sinusfunksjoner med frekvenser lik et helt antall ganger frekvensen i den periodiske funksjonen. SymPy har ikke funksjoner som gjør dette på en enkel måte, men tar i det minste grovarbeidet.



Figuren viser en sagtannkurve med periode $P = 2L = 8$ som plot av den harmoniske tidsfunksjonen

$$f(t) = \frac{t}{2} - 1 \ , \ -4 < t <= 4 \ , \ f(t+8) = f(t)$$

Fourierrekka er summen av middelverdi, $\frac{1}{2}a_0$, cosinusledd med amplituder a_n og sinusledd med amplituder b_n ,

$$\frac{1}{2}a_0 + \sum_{n=1}^{\infty} a_n \cos(\frac{n\pi}{L}t) + \sum_{n=1}^{\infty} b_n \sin(\frac{n\pi}{L}t)$$

- der amplitudefaktorene beregnes slik:

$$a_0 = \frac{1}{L}\int_{-L}^{L} f(t)\,dt = \frac{1}{L}\int_{-L}^{L} \frac{t}{2} - 1\,dt$$

$$a_n = \frac{1}{L}\int_{-L}^{L} f(t)\cdot\cos(\frac{n\pi}{L}t)\,dt = \frac{1}{L}\int_{-L}^{L}(\frac{t}{2} - 1)\cdot\cos\frac{n\pi}{L}t)\,dt$$

$$b_n = \frac{1}{L}\int_{-L}^{L} f(t)\cdot\sin(\frac{n\pi}{L}t)\,dt = \frac{1}{L}\int_{-L}^{L}(\frac{t}{2} - 1)\cdot\sin(\frac{n\pi}{L}t)\,dt$$

```
>>> t = symbols('t')
>>> n = symbols('n', integer=True, positive=True)
>>> f = t/2 - 1
>>> L = S('4')
>>> a0 = 1/L*integrate(f, [t, -L, L])
>>> a0
-2
>>> an = 1/L*integrate(f*cos(n*pi/L*t), [t, -L, L])
>>> an
0
>>> bn = 1/L*integrate(f*sin(n*pi/L*t), [t, -L, L])
>>> bn
-4*(-1)**n/(pi*n)
>>> [an.subs(n, i) for i in range(1, 8)]
[0, 0, 0, 0, 0, 0, 0]
>>> [bn.subs(n, i) for i in range(1, 8)]
[4/pi, -2/pi, 4/(3*pi), -1/pi, 4/(5*pi), -2/(3*pi), 4/(7*pi)]
```

De første linjene til forbereder beregningene,
- setter opp t (tid) som symbolsk variabel
- setter opp heltallet n med positive bare verdier
- lar halvperioden på 4 være en symbolsk verdi
- legger tidsfunksjonen i variabelen f

-deretter beregnes a_0 til verdien -2 (middelverdi=1/2 a_0 = -1) som vi også ser av kurven og funksjonsuttrykket. Integralene som gir amplitudene a_n og b_n beregnes med heltallet n som variabel. Hvis

det ikke ble satt tallområde for n ville SymPy ha integrert som om n var et reelt tall - og gitt resultater i forhold til det.

Det som nå står igjen er å tolke resultatene slik at vi kan sette opp en generell formel for a_n og b_n. Sinusleddene har amplituder lik null, $a_n = 0$, men cosinusamplitudene virker i første omgang litt uryddige. Med litt fantasi ser vi at de kan skrives som

$$\{\frac{4}{\pi}, -\frac{4}{2\pi}, \frac{4}{3\pi}, -\frac{4}{4\pi}, \frac{4}{5\pi}, -\frac{4}{6\pi}, \frac{4}{7\pi}\} = \frac{4}{\pi}\{\frac{1}{1}, -\frac{1}{2}, \frac{1}{3}, -\frac{1}{4}, \frac{1}{5}, -\frac{1}{6}, \frac{1}{7}\}$$

som er en alternerende tallfølge av typen $\{\frac{4}{\pi}(-1)^{n+1}\frac{1}{n}\}$.

Fourierrekka med konstant $\frac{1}{2}a_0$ og de 7 første leddene får vi ved å multiplisere konstantene i vektorene an og bn med cos(nt) og sin(nt) og samle alt til en sum, $F = -1 + \frac{4}{\pi}\sum_{n=1}^{\infty}(-1)^{n-1}\frac{1}{n} \cdot \sin(\frac{n\pi}{L}t)$. Så plotter vi fourierrekka med de 7 første leddene:

```
>>> F7 = a0/2 + summation(an*cos(n*pi/L*t) + bn*sin(n*pi/L*t), [n, 1, 7])
>>> F7
4*sin(pi*t/4)/pi - 2*sin(pi*t/2)/pi + 4*sin(3*pi*t/4)/(3*pi)
- sin(pi*t)/pi + 4*sin(5*pi*t/4)/(5*pi) - 2*sin(3*pi*t/2)/(3*pi)
+ 4*sin(7*pi*t/4)/(7*pi) - 1
>>> plot(F7)
```

Logikk og mengder

Logikk

```
True, true          Logisk SANN
False, false        Logisk FALSK
~, not              Logisk IKKE
&, and              Logisk OG
|, or               Logisk ELLER
>>                  Logisk IMPLIKASJON
simplify_logic      Forenkler logiske uttrykk
```

Python utfører logiske operasjoner som IKKE, OG og ELLER litt forskjellig fra logikken du kjenner fra diskret matematikk. Det henger sammen med at Python er et generelt programmeringsspråk som gjør bitvise logiske operasjoner på tall generelt. Sannhetsverdiene i Python er `False` og `True`. Med SymPy importert kan vi gjøre logiske operasjoner med sannhetsverdiene `false` og `true` (med liten forbokstav) som vi kjenner fra mattebøkene.

Logikk er basert på de to sannhetsverdiene `false` og `true` eller alternativt som 0 eller 1. Logiske operatorer kombinerer sannhetsverdier til sammensatte uttrykk - boolsk algebra.

```
Logisk IKKE      operator  ~  alternativt  not
Logisk OG        operator  &  alternativt  and
Logisk ELLER     operator  |  alternativt  or
```

Resultatet av logiske operasjoner beskrives enklest i sannhetstabeller (F=false og T=true):

```
 IKKE       _____OG_____      ____ELLER____
a  ~a       a b    a&b        a b    a|b
F   T       F F     F         F F     F
T   F       F T     F         F T     T
            T F     F         T F     T
            T T     T         T T     T
```

Logiske operatorasjoner kan enten settes opp med operatortegn eller som funksjoner, Not(a), And(a, b), Or(a, b), Xor(a, b). Funksjonen Xor(a, b) utfører eksklusiv eller, Xor(0,0)=0, Xor(0,1)=1, Xor(1,0)=1 og Xor(1,1)=0.

```
>>> from sympy import *                  # nå kan vi bruke false og true
>>> false | true & ~false
True
>>> false and true or  not false
True
>>> 0 | 1 & ~0
1                                        # Hvorfor ikke True?
>>> Or(false, And(true, Not(false)))
True
>>> kino = sponsing and not lekser
>>> kino
False
```

Sannhetsverdiene i SymPy kan tastes inn som `false` eller `true` eller som tallstørrelser 0, 1. Svarene leveres som `False` eller `True` (med Stor forbokstav). De logiske operatorene har presedens etter ordningen

```
~   not    Logisk IKKE
&   and    Logisk OG
|   or     Logisk ELLER
```

slik at uttrykket `a or not b and c` evalueres som or(a, and(not(b), c)), altså først negering av b, deretter og-operasjonen og til slutt eller-operasjonen.

Logisk implikasjon a→b bruker >> som operator, `a >> b`.

```
>>> F = false; T = true                  # Lager kortformer
>>> F >> F; F >> T; T >> F; F >> F       # Tester logisk implikasjon
True
True
False
True
```

En av SymPys hyggeligere logiske funksjoner er `simplify_logic` som gjør forenklinger i et sammensatt logisk uttrykk. La oss prøve med `b = (~x & ~y & ~z) | ( ~x & ~y & z)`,

```
>>> x, y, z = symbols('x y z')
>>> simplify_logic((~x & ~y & ~z) | ( ~x & ~y & z))
And(Not(x), Not(y))
```

Hvis du er litt dreven i å lese enlinje uttrykk, vil du se at dette tilsvarer `~x & ~y`.

Med litt løkkeprogrammering kan vi lage sannhetstabeller ut fra logiske uttrykk. Her er det vist hvordan sannhetstabellen for logisk implikasjon ser ut med p og q som inngangsverdier. Sjekk med din yndlingsmattebok.

```
>>> for x in [F, T]:
        for y in [F, T]:
                s = '{0:8s}{1:8s}{2:12s}'
                print(s.format(repr(x), repr(y), repr(x>>y)))

False   False      True
False   True       True
True    False      False
True    True       True
```

Mengder

```
|              Unionoperator
&              Snittoperator
-              Mengdedifferanse
in             Tester element i mengde
issubset       Tester for delmengde
```

Python har en egen datatype som lagrer mengder (set). Elementene i en mengde kan være tall, bokstavtegn, tekststrenger, og andre sammensatte datatyper. Vi setter opp en mengde mellom et par krøllparenteser.

```
>>> mengde1 = {'a', 'b', 'c'}
>>> mengde2 = {'e', 'b', 'f', 'a'}
>>> 'a' in mengde1                      # Er 'a' element i mengde1?
True
>>> 'c' in mengde2
False
```

Mengdeoperasjonene snitt, union og mengdedifferanse bruker samme operatorer som logisk ELLER, logisk OG og minustegn,

```
>>> mengde1 | mengde2                  # Unionen av mengdene
{'c', 'a', 'f', 'e', 'b'}
>>> mengde2 & mengde1                  # snittet av mengdene
{'a', 'b'}
>>> mengde1 - mengde2                  # Mengdedifferanse, de elementene som
{'c'}                                  # er i 'mengde1', men ikke i 'mengde2'
>>> mengde2 - mengde1
{'e', 'f'}
```

Vi kan finne ut om en mengde er delmengde i en annen mengde med metoden `.issubset`,

```
>>> {'b', 'c'}.issubset(mengde1)
True
>>> {'b', 'c'}.issubset(mengde2)
False
```

Matriser og vektorer

Matriser

```
Matrix          Lager matriser
det             Beregner determinant
inv, **(-1)     Finner invers matrise
rref            Radreduksjon (row reduced echelon form)
```

En matrise er rett og slett en ordnet stabel med tall eller andre datatyper. I faget matematikk er matriser byggesteinene i emnet lineær algebra med teoremer og teknikker for å lage sum, produkt og andre operasjoner med matriser. Slå opp i din yndlingsmattebok om du er usikker på definisjonene.

SymPy Lager matriser med funksjonen `Matrix`, der rader og kolonner settes opp innenfor hakeparenteser,

```
>>> A = Matrix([[3, -2], [5,  1]])     # En 2x2 matrise
>>> A
Matrix([
[3, -2],
[5,  1]])
>>> B = Matrix([[0, 1], [-2, 0]])
```

```
>>> B
Matrix([
[ 0, 1],
[-2, 0]])
>>> A+B                                  # Sum av matriser
Matrix([
[3, -1],
[3,  1]])
>>> A*B                                  # Matriseprodukt
Matrix([
[ 4, 3],
[-2, 5]])
>>> B*A                                  # Matriseproduktet
Matrix([                                 # er ikke kommutativt,
[ 5, 1],                                 # A*B er forskjellig fra B*A
[-6, 4]])
```

Determinanten til en kvadratisk matrise beregnes med funksjonen `det`. Den inverse av en matrise finnes med metodenen `inv` eller eksponentoperasjon. Reduksjon av matriser skjer med funksjonene `jordan_form` og `rref`.

I lineær algebra løser vi likningssett med matrisemanipulering. Vi setter opp en matrise A for koeffisientene og en matrise B for konstantene i likningen

$$\begin{aligned} 2x + y &= 5 \\ x - 2y &= -5 \end{aligned}$$

```
>>> from sympy import *
>>> A = Matrix([[2, 1], [1, -2]])       # Koeffisientmatrise
>>> B = Matrix([[4], [-5]])             # Konstantmatrise
```

Et slikt likningssett har entydig løsning dersom determinanten til A er forskjellig fra null,

```
>>> det(A)
-5
```

Bra, det stemmer. Teorien sier nå at vi kan finne løsningene som produktet av den inverse matrisa til A og matrise B,

```
>>> A.inv()*B                           # Matrisemultiplikasjon, A^-1×B
Matrix([
[1],                                    # x=1
[3]])                                   # y=3
```

En alternativ måte å løse et lineært likningssett er å sette opp en totalmatrise og gjøre forenklinger ved hjelp av matrisealgebra. Ved å redusere matrisa til Jordan (reduced row echelon form) form vil vi ende opp med et ekvivalent likningssett der løsningene er enkle å finne.

```
>>> L = Matrix([[2, 1, 5], [1, -2, -5]])
>>> L
Matrix([
[2,  1,  5],
[1, -2, -5]])
>>> L.rref()
(Matrix([
[1, 0, 1],
[0, 1, 3]]), [0, 1])
```

De to matrisene `L` og `L.rref()` er ekvivalente, de har samme løsning. Den første linja i siste matrise tolker vi som '$1x + 0y = 1$' og andre linje som '$0x + 1y = 3$', altså løsning x=1 og y=3. Den andre lista, `[0, 1]`, gir opplysning om at innledende 1-ere i den reduserte formen er på posisjonene 0 og 1.

Nå har vi altså sett flere måter å løse slike likninger på. Vi tar en siste sjekk med universalredskapen `solve`,

```
>>> solve([2*x+y-5, x-2*y+5])
{x: 1, y: 3}
```

Vedlegg: Litt Python

Se også view-sourcePython på 4 minutter

Her er en ultrakort innføring i Python. Hensikten er å presentere datatyper, moduler og litt programmeringsteknikk.

Datatyper

Tall er enten heltall, flyttall (desimaltall) eller komplekse tall. Regneuttrykk med heltall og flyttall blir flyttall.

```
>>> 1 + 2 - 3 * 4 / 5                  # Divisjonen gir flyttall (Python3)
0.6000000000000001
>>> 1 + 2 - 3 * 4 // 5                 # Heltallsdivisjon (Python3)
1
```

Stringer, lister, tupler (tuples) og oppslagslister (dictionaries) er sekvensielle datatype som lagrer verdier i tabellform. Stringer og tupler defineres med dataverdier som ikke kan endres. Indekseringene starter med 0 som første element.

```
>>> navn = 'Abraham'                          # string
>>> oddetall = [1, 3, 5, 7]                   # list
>>> nevoer = ('Ole', 'Dole', 'Doffen')        # tuple
                                              # dictionary:
>>> persondata = {'fornavn': 'Donald', 'etternavn': 'Duck', 'alder': 78}
>>> navn[0]
'A'
>>> navn[0]= 'B'
#...
TypeError: 'str' object does not support item assignment
>>> oddetall[2]= 9
>>> oddetall
[1, 3, 9, 7]
nevoer[1] = 'Petter'
#...
TypeError: 'tuple' object does not support item assignment
>>> persondata['alder'] = 89
>>> persondata
{'alder': 89, 'fornavn': 'Donald', 'etternavn': 'Duck'}
```

Legg merke til bruken av de forskjellige parentestypene. Hakeparenteser brukes ved indeksering. I matematikk vil funksjoner som finner flere løsninger gi disse i en liste.

Moduler, navneområder

Moduler er samlinger av Python kode som utfører spesialiserte oppgaver. Modulene inneholder definisjoner av variabler, konstanter, funksjoner og klasser. Noen moduler, som math og cmath, følger

med i standard installasjon av Python, andre må installeres separat, for eksempel SymPy. Når vi skal ta i bruk en modul må vi enten importere den som et eget navneområde (namespace), eller legge den inn i det gjeldende navneområdet.

```
>>> sqrt(8)                             # Funksjonen sqrt er ikke med i standard Python
#...
NameError: name 'sqrt' is not defined
>>> import math                         # Leser inn modulen math
>>> math.sqrt(8)                        # Bruker sqrt fra navneområdet math
2.8284271247461903
>>> from math import *                  # Lar math bli med i gjeldende navneområde
>>> sqrt(8)
2.8284271247461903
>>> from sympy import *                 # Leser inn sympy til gjeldende navneområde
>>> sqrt(8)                             # Bruker sqrt fra sympy
2*sqrt(2)
```

Symbolsk matematikk hentes fra modulen `sympy`. Symbolske regneuttrykk må lages som stringer til funksjonen `S` eller med symbolske variable.

```
>>> from sympy import *
>>> S('1/2 + 1/3')                      # S = sympify
5/6
>>> x = symbols ('x')                   # definerer symbolsk variabel
>>> 6*x-5-4*x+3
2*x - 2
```

Funksjoner, klasser og objekter

En funksjon er et stykke Python kode som utfører en avgrenset oppgave, for eksempel å finne kvadratrota av et tall. Når vi bruker funksjonen gjør vi et funksjonskall av formen *`funksjonsnavn`*`(`*`inngangsverdi`*`)`, der vi gjerne sender med en (eller flere) inngangsverdier. Funksjonen lager et resultat og sender det tilbake til programkjøringen i form av en returverdi.

```
>>> def rot(x):                         # Dette er Python kode
        return x**(1/2)                 # som definerer en funksjon

>>> rot(4)                              # Her er et funksjonskall
2.0                                     # som gir denne returverdien
```

Funksjonen rot som ble laget ovenfor eksisterer bare så lenge Pythonskallet kjører, men om linjene legges i en fil som lagres med filnavn `navn.py`, vil vi kunne bruke funksjonen senere ved å importere `navn` som en modul.

En klasse (class) er en samling funksjoner og variabler som beskriver en datatype. Vi har allerede møtt klassene heltall, komplekse tall, stringer, lister og andre. Et objekt er en enkelt instans (individ) av en klasse, på samme måte som at 'Tja' er et objekt av klassen string.

I modulene ligger det klassedefinisjoner som naturlig hører sammen. Vi henter et eksempel på modulen `cmath` som håndterer matematikk med komplekse objekter (tall). La oss sette opp to tall, `w` og `z`, og gjøre noen sjongleringer.

Noen datatyper og operasjoner med disse er altså innebygget i Python. Når vi bruker Python som kalkulator blir det ikke så mye snakk om klasser og objekter. La oss sette opp to komplekse tall, `w` og `z`, og gjøre noen sjongleringer.

```
>>> w = complex(3, -4)                  # Vi 'konstruerer' objektet w av klassen complex
>>> z = 2 + 1j                          # Objektet z konstrueres med kortform
>>> w+z                                 # Enkel matematikk fungerer
```

```
(5-3j)
>>> z.imag                              # Henter ut imaginærverdien av z
1
>>> w.conjugate()                       # Bruker klassemetoden conjugate() til å hente
konjugert verdi av w
(3+4j)
>>> >>> z*w                                  # Men, w er ikke endret
(3-4j)
>>> w
(3-4j)
```

Her var det mye på en gang. Selve klassenavnet brukes som konstruktør, objektet `w` av klassen `complex` settes opp (konstrueres) i linja `w = complex(3, 4)` med komponentene 3 og 4 som innverdier. Vi ser at komplekse tall er innebygget som klasse i Python. Derfor er det også laget en mulighet til å taste inn et komplekst tall på kortform, `z=2+1j`, som alternativ til formell konstrutør. Det samme gjelder også tall, der `7` blir et objekt av heltallsklassen, mens `7.0` blir et objekt av float-klassen.

Et objekt har gjerne dataverdier som ligger lagret i attributter. Det har også funksjoner som gjør det i stand til å sjonglere med attributtene. Imaginærverdien av `z` kan vi finne ved å legge .imag bak objektnavnet, `z.imag`. Den konjugerte verdien av `w` kan vi finne ved å tilkalle metoden (funksjonen) conjugate, `w.conjugate()`. Legg merke til forskjellene her, `z.imag` henter ut en dataverdi fra objektet, mens `w.conjugate()` gjør en operasjon på objektet. Til slutt bruker vi *funksjonen* `abs` til å finne absoluttverdien (lengden) av `w`. Denne funksjonen er altså laget slik at den kan operere med objekter.

Vi går litt videre og onsker å finne kavdratrota av `z`, og prøver med funksjonen `sqrt`,

```
>>> sqrt(w)
Traceback (most recent call last):
  File "<pyshell#100>", line 1, in <module>
    sqrt(w)
NameError: name 'sqrt' is not defined
```

Neivel, kanskje ble det litt for avansert for standard Python, la oss importere modulen `cmath`,

```
>>> import cmath
>>> cmath.sqrt(w)                       # Bruker funksjonen sqrt i modulen cmath
(2+1j)
>>> _**2                                # Tester siste svar om vi virkelig fant
kvadratrot
(3+4j)
```

Her brukes funksjonen `sqrt` som er definert i modulen `cmath` til å beregne kvadratrota av det komplekse tallet `w`. Vi må altså henvise til funksjoner i en importert modul med '`modulnavn.`'. En annen måte å gjøre det samme på er altså å importere alle funksjoner i `cmath` inn i det gjeldende navneområdet. Da bruker vi funksjonsnavnene direkte,

```
>>> from cmath import *
>>> sqrt(w)
(2+1j)
>>> z.conjugate().imag * sqrt(w)
(-2-1j)
```