Forord

Dette heftet er ment som en støtte til undervisningen i forkurset i informatikk for MENA og LAP. Hovedvekten vil ligge på språket Python. Håpet er at denne teksten er realtivt komprimert og grei å lese.

Foreløpig er teksten noe uferdig, og dersom du mistenker en feil, har du mest sannsynlig funnet en. Feilrapporter mottas med stor takk av forfatteren.

Stor takk til Hans Petter Langtangen, Aslak Tveito, Ilmar Wilbers og Morten Hjorth-Jensen for utlån av massevis av stoff! Fra kursheftet i INF1100 har jeg lånt, og hentet både eksempler, og latt meg inspirere av teksten. Avsnittene om differensiallikninger er så og si en blåkopi av Morten Hjorth-Jensens forelesningsnotater for Computational Physics 1s behandling av temaet.

Det er viktig å huske at ingen kan lærer seg å programmere ved å lese, akkurat som at ingen har lært seg et språk ved å lese en grammatikkbok. Du er nødt til å skrive mange (nokså tåpelige) programmer selv. Gjør oppgavene til forkurset, og gjerne også flere oppgaver for eksempel fra pensumheftet til INF1100.

Kursheftet for INF1100 anbefales på det sterkeste som tilleggslesning, og for videre arbeid med temaene som ikke er behandlet i dette forkurset. På hjemmesiden til INF1100 (http://www.uio.no/studier/emner/matnat/ifi/INF1100/h08/) finnes også lenker til mange andre læringsressurser.

En av disse er Sindre Froyns flashfilmer om python som du finner på http://folk.uio.no/sindrf/python/. Se disse i forbindelse med stoffet som gjennomgåes. Før eller etter forelesning, eller for å sjekke noe du står fast på mens du programemrerer.

For videre arbeid og studier av metoder anbefales Morten Hjorth-Jensens hefte til FYS3150, og også selvsagt kurset FYS3150.

Oslo 24.07.2008 Marit Sandstad

Chapter 1

Dag 1: Det grunnleggende

I dette kapittelet skal vi gjennomgå litt om hva programmering er. Hvordan vi kommer igang med å programmere i python, og lære litt om endel grunnleggende konstuksjoner i språket.

1.1 Hva er programmering, hva er et program?

Programmering er en måte å instruere datamaskinen om å gjøre forskjellige ting. Det kan være å skrive ut en setning, gjøre en beregning, eller å gjøre mange forskjellige ting etter hverandre til en avansert struktur, slik at du får f.eks et operativsystem.

Som et eksempel kan vi vise hvordan vi skriver ut svaret fra en utregning. Vi velger å se på høyden til en ball med masse $m = 0,6$ som blir kastet oppover med en startfart $v_0 = 5m/s$ ved tiden $t = 0.6s$ etter at ballen er blitt kastet. Vi vet at tyngdeakselerasjonen er $9.81m/s^2$ og den generelle formelen:

$$y(t) = v_0 t - \frac{1}{2} g t^2 \text{gir :} \tag{1.1}$$

$$y(0.6) = 5 * 0.6 - \frac{1}{2} * 9.81 * 0.6^2$$

For å regne ut dette i python og skrive ut resultatet til skjermen må vi skrive:

```
print 5*0.6 - 0.5*9.81*0.6**2
```

I dette kurset vil vi lære enkel programmering for å gjøre beregninger i programmeringsspråket python. Et programmeringsspråk kan vi tenke på

som et språk som datamaskinen forstår [1]. Å programmere blir da en måte å snakke med datamaskinen på. Det finnes mange forskjellige programmeringsspråk med ulike fordeler og ulemper. Programmeringsspråk har som regel en relativt enkel syntaks [2], endel reserverte spesialord, og en enkel grammatikk. I python er følgende ord reservert:

```
and      as      assert   break   class   continue
def      del     elif     else    except  exec
finally  for     from     global  if      import
in       is      lambda   not     or      pass
print    raise   return   try     with    while
yield
```

Den største forskjellen på programeringsspråk og vanlige språk er kanskje hvor nøyaktig man må være. Datamaskinen er en ekstrem pedant. Hvis du skriver noe som likner svært mye på det du mente, men med en eller annen feil, vil datamaskinen enten ikke forstå hva du har ment og programmet "kræsjer", eller enda verre, den vil gjøre noe annet enn det du mente. Vi skal gå nærmere inn på feil, feilmeldinger, feilsøking og kræsj senere, og du vil selv få erfare dette når du programmerer.

1.2 Å komme i gang. Hvordan kan vi skrive programmer?

Det finnes to måter å skrive programmer i python på.

Den første, som passer best for litt større programmer, og som er den eneste måten å skrive endel andre språk, som Java, C/C++ og Fortran på, er ved å lage en tekstfil med hele programmet. Her skriver du inn alt du vil at datamaskinen skal gjøre etter hverandre i en rekkefølge. Deretter kjører du programmet. For å lage denne tekstfilen bruker man en editor.

I prinsippet kan vi bruke et hvilketsomhelst tekstbehandlingsprogram, men det finnes editorer som er spesiallaget for programmering. Da får du blant annet farger på teksten din som svarer til ulike aspekter ved språket, og automatisk innrykk. Begge deler er nyttig når vi programmerer. Det første fordi programmet blir lettere å lese, og det blir lettere å finne feil. Det andre er særlig fint i python, for i python er innrykket viktig for betydningen av det vi har skrevet.

I dette kurset vil jeg vise eksempler med tekstbehandleren emacs.

[1] for de fleste programmeringsspråk er det strengt tatt ikke slik at datamaskinen forstår det. Derimot finnes det et tolkeprogram kalt en kompilator, som oversetter språket til sekvenser av 0-er og 1-ere, eller binærkode. Egentlig er denne binærkoden det eneste datamaskinen forstår

[2] Syntaks betyr noe sånt som setningsbygning og konstruksjon. Norsk syntaks innebærer for eksempel at verbalet skal være det andre leddet i en setning, dersom det ikke er et spørsmål. Dersom vi skriver "Spiser jeg grøt", må vi altså avslutte med spørsmålstegn hvis det skal være en lovlig konstruksjon på norsk.

Den andre måte å skrive programmer i python på, er å starte et interaktivt programmeringsmiljø. Det vil si at du starter et program der du kan skrive hver kommando i python-programmet, og kjøre dem direkte. I dette kurset vil du bli kjent med programmet IPython for å gjøre dette. Å kjøre hver linje i en interaktiv økt (eng: interactive session), er den vanligste måten å bruke programmeringsspråk som Matlab på.

Uansett hvilken måte vi velger å skrive programmet vårt på, må vi starte med et terminalvindu. Et terminalvindu, eller kommandolinjevindu, er et vindu der vi kan skrive inn kommandoer til datamaskinen. En liste med noen nyttige kommandoer er å finne i et vedlegg til denne teksten. For å starte IPython skriver du:

```
ipython
```

Så er det bare å begynne.

For å skrive et program i emacs, åpner du først emacs ved å skrive:

```
emacs &
```

&-tegnet er ikke nødvendig, men gjør det mulig å fortsette å bruke kommandolinjevinduet, uten å lukke emacs igjen. Du får opp en blankt emacsvindu med informasjon om emacs. Dersom du trykker på det blanke arket i venstre hjørnet, får du opp et tilnærmet blankt ark, og beskjed om å velge filen du vil åpne nederst. Du kan begynne å skrive uten å ha et navn, men det greieste er å velge et filnavn med en gang. Velg deg et navn som sier noe om hva programmet skal gjøre, og avslutt navnet med ".py".Når du har skrevet inn filnavnet er det bare å begynne å skrive programmet. Når programmet begynner å bli ferdig (eller ihvertfall kjørbart), kan du prøve å kjøre programmet. Det gjør du ved å gå tilbake til terminalvinduet og skrive:

```
python program.py
```

(Du bytter selvsagt ut "program.py" med navnet på programmet du vil kjøre). Du kan også kjøre et program du har laget i en editor i en interaktiv økt i IPython. Du skriver da:

```
run program.py
```

Dersom det ikke er feil i programmet, vil datamaskinen da gjøre det du har programmert den til å gjøre. Som regel vil du imidlertid få en feilmelding de første gangene du prøver dette. Det er ingen grunn til å få panikk når dette skjer. I feilmeldingen står det litt om hva slags feil du har fått, og på hvilken linje feilen har oppstått. De vanligste feiltypene er beskrevet i neste kapittel.

1.3 Gjenbruk av størrelser: Variable

Dersom vi skal skrive programmer som gjør litt mer enn rene kalkulatorutregninger, trenger vi å kalle størrelsene noe slik at vi kan bruke dem igjen. Dette er variable. Det finnes ulike typer variable, og hvilke som finnes er avhengig av språket du skriver i. I språk som Java må du deklarere hva slags variabel som kommer før du skriver dem (d.v.s. at du sier eksplisitt hvilken variabeltype den skal ha når du gir den navn). I python er dette ikke nødvendig, kompilatoren, eller datamaskinen, forstår hva slags variabeltype du skal ha på måten du initialiserer den på (å initialisere vil si å sette den til en verdi). At du ikke må deklarere variabeltypen gjør at koden blir kortere, men det finnes også ulemper. Vi kommer nærmere inn på dem etterhvert.

I python er det mange innebygde variabeltyper. Under lister vi opp variabeltypene vi skal bruke. Det går også an å definere sine egne variabeltyper, ved hjelp av noe som kalles klasser og objektorientering. Vi skal ikke gå nærmere innpå det her.

Variabeltypene som er viktige for oss er:

1.3.1 int

Type for heltall. Vi initialiserer en variabel til å være en int ved en setning (eng: statement) av denne typen:

```
masse = 20
```

1.3.2 float

Type for flyttall, eller desimaltall. Vi initialiserer en float ved en setning av følgende type:

```
masse = 20.0
```

1.3.3 complex

Type for komplekse tall. Disse settes for eksempel på følgende måte:

```
ctall = 4.0 + 6.48j
```

For komplekse tall, kan du skrive ut både hele tallet, og realverdien og imaginærverdien for seg. De enkelte verdiene får du tak i ved å kalle på ctall.real, ctall.imag.

1.3.4 bool

Type for boolske variable. Boolske variable er sannhetsverdier. Vi kan sette en slik sannhetsverdi ved bare å skrive inn verdien True eller False direkte, eller vi kan sette den til evaluering av et sannhetsuttrykk. I sannhetsuttrykk sammenlikner vi ofte to verdier ved å se om de er lik hverandre, om den ene er større enn den andre o.s.v. De vanligste tegnene for slik sammenlikninger finner du i tabellen under:

Tegn	Betydning	Tegn	Betydning
==	Er lik	!=	Er forskjellig fra
<	Mindre enn	<=	Mindre eller lik
>	Større enn	>=	Større eller lik

At variabelverdier settes ved evaluering av uttrykk gjelder selvsagt også ved andre variabeltyper. Når det gjelder tallverdier gjøres dette ved f.eks. de aritmetiske operasjonene.

Bruken av boolske variable skal vi komme nærmere innpå i avsnittet om if-forgreninger og løkker under. Her er tre eksempler på hvordan du kan initialisere boolske variable:

```
bolsk1 = True
bolsk2 = False
bolsk3 = (bolsk1 == bolsk2)
```

1.3.5 str eller String

Typer for tekststrenger. Tekststrenger er noe vi kaller uforanderlige (eng: immutable). Det betyr at de i prinsippet ikke kan endres. Variabeltyper som ikke er immutable, kan settes til nye verdier, og vi beholder da ikke den gamle verdien. Når vi endrer en uforanderlig variabel oppretter vi i prinsippet en helt ny en som heter det samme. Det vil si at den gamle verdien blir liggende i minnet, men siden den nye har overtatt navnet dens, har vi ingen måte å få tak i den på. Som regel er dette ikke så viktig, men dersom man trenger fart og minneplass, kan det være noe å tenke over.

En str, eller tekststreng initialiseres:

```
tekst1 = "kakedeig"
tekst2 = 'deigkake'
```

Legg merke til at både enkelt og dobbelt anførselstegn er lov når man skiller ut tekststrenger. Det viktigste er at man begynner å avslutter med samme type anførselstegn.

1.3.6 tuple

Et tuppel er en rekke med verdier av forskjellige typer i en parates og separert med komma. Som str er den uforanderlig. Vi intitialiserer et tuppel på følgende måte:

```
litt_forskjellig = ("Ball", "masse", 20.0, "kg", True)
```

For å få tilgang til en enkelt verdi i tuppelet, kan vi aksessere det ved å utnytte indeksering av verdiene. Verdiene i et tuppel har en tilhørende indeks fra 0 til n-1, der n er antallet verdier i tuppelet. For å få ut 20.0 fra tuppelet over, må vi derfor skrive litt_forskjellig[2]. Vi kan også få tak i elementene ved å telle indeksering bakfra. Det siste elementet har da indeks -1, det nest siste -2, o.s.v. Vi kan også få tak i lengre sammenhengende deler av tuppelet. For eksempel kan vi få et tuppel med alle de samme elementene som det opprinnelige tuppelet fra element nummer p til element nummer q. Dette får vi tak i ved å skrive litt_forskjellig[p:q+1].

Det hender også ofte at vi vil vite lengden på et tuppel. Denne lengden, altså tallet n på antall elementer, får vi ut ved å be om len(tuppelet). Hvis vi vil få ut dette tallet i eksempelet over, kan vi for eksempel skrive: n = len(litt_forskjellig)

Tupler bruker vi for det meste til å sende rekker av verdier rundt i programmet. Vi bruker det også til formatert utskrift. Dersom vi skal skrive ut noe, kan vi nemlig lage en tekststreng der vi setter av plass til verdier av ulike typer. Hver slik er markert med et prosenttegn, etterfulgt av noen bokstaver som angir typen variabel vi vil sette inn. Etter at tekststrengen med slike plasser avsatt er fullført, følger et nytt prosenttegn, og deretter et tuppel som inneholder de verdiene vi vil putte inn, i den rekkefølgen vi vil ha dem i sekvensen. Vi kan også putte variable vi har opprettet tidligere, inn i et slikt tuppel.

En liste over avmerkingene, eller formatene vi bruker inne i teksstrengen følger under:

```
%s              en tekststreng (string/str)
%d              et heltall (int/integer)
%0xd            et heltall med x ledende nuller foran (bytt x med et tall)
%f              et flyttall notert med 6 desimaler
%e              kompakt vitenskapelig notasjon med e som eksponent.
%E              kompakt vitenskapelig notasjon med E som eksponent
%g              kompakt desimal eller vitenskapelig notasjon, e som eksponent
%G              kompakt desimal eller vitenskapelig notasjon, E som eksponent
%xz             formatet z tilpasset til et felt med bredde på x tegn
%-xz            formatet z, venstrejustert til et felt med bredde på x tegn
%.yz            formatet z med y desimaler
%x.yz           formatet z med y desimaler tilpasset til et felt med bredde x
%%              et prosenttegn
```

Vi gir nå et eksempel på en utskrift av en formatert tekststreng:

```
masse = 20.0
print "%d baller veier %f kg. Det er tyngre enn en %s" \
       % (2, masse, "golfball")
```

Tegnet \ på første linje må til for at de to linjene skal bli som en.

1.3.7 Lister - list

En liste er den foranderlige tvillingbroren til et tuppel. (En liste er altså ikke immutable.). Både lister og tupler kan ha elementer av alle typer, begge kan ha vilkårlig lengde, og vi får fatt på hvert element i listen på samme måte. Rent utseendemessig består forskjellen i at lister har firkantede parenteser rundt. Siden listen er foranderlig, har lister likevel et ganske ulikt bruksområde enn tupler. Vi itererer ofte over lister (vi kan gjøre dette med tupler også, men det er mer fordelaktig å bruke lister fordi de er foranderlige), vi skal gå nærmere innpå dette når vi snakker om løkker. De ulike elementene i en liste står oftere i et forhold til hverandre.

En liste initialiserer vi på følgende måte:

```
liste = ["a", "b", "c", 4]
```

Siden verdiene i en liste (og forsåvidt også i et tuppel) kan være av hvilken type vi vil, kan vi også lage lister av lister. Slike lister kalles nestede lister eller tabeller. Vi skal utforske dette litt nærmere i programmeringsoppgavene, men gir et enkelt eksempel her:

```
liste = [["H", "He"],["Li", "Be", "B", "C", "N", "O", "F", "Ne"],
         ["Na" ,"Mg", "Al", "Si", "P", "S", "Cl", "Ar"]]
print liste[0], liste[1][3]
```

Her består listen av tre lister, hver bestående av en periode fra det periodiske system. Utskriften gir da først den første perioden, så grunnstoffet C i den andre perioden.

Siden listen er foranderlig, kan vi også legge til nye elementer i listen eller slette gamle elementer. Vi legger til et nytt element til slutt i listen ved å skrive liste.append(nyttelement). Vi sletter for eksempel det første elementet ved å skrive del liste[0].

[3]

Det finnes også noen spesielle ferdiglagede lister som er spesielt hendige. Noen ganger vil vi for eksempel ha en liste med mange like elementer. La oss si at vi ønsker oss en liste med 10 nuller. Vi skriver da liste = [0]*10. Det

[3] Legg merke til at i eksempelet over trenger vi ikke \ for å vise at innholdet i listen fortsetter på neste linje. Dette er fordi den første parentesen ([) ikke er avsluttet når vi kommer til slutten av linja. kompilatoren vil fortsette å lese programmet som om det stod på en linje, helt til denne parentesen er avsluttet. Dette kan føre til at dersom du glemmer å lukke en parentes i programmet ditt, får du en feilmelding om dette som sier at parentesen ikke er avsluttet først helt på slutten av programmet.

hender også at vi vil ha en liste med tallene fra et tall til et annet. Dette får vi fra noe som heter range. range(m,n,k) gir en liste med laveste element m, høyeste element så stort som mulig, men lavere enn n og med hopp mellom elementene på k. (m,n og k må være heltall). Vi kan utelate k, og får da hopp på 1. Dersom vi bare skriver range(n) får vi en liste med tallene 0 opptil $n-1$.

1.3.8 Dictionaries, ordbøker, dict

Dictionaries, eller ordbøker, er kort fortalt lister der listen ikke er indeksert ved tall fra 0 til n-1, men hvert element er indeksert ved et annet element. Du får tak i hver verdi (eng: value) ved hjelp av en tilhørende nøkkel (eng: key). Dictionaries kan være svært nyttige, men vi skal ikke gå så nøye innpå dem her. Et eksempel på initialisering av en dictionary er:

```
grunnstoffvekt = {"H":1.00794, "He":4.002602, "Li": 6.941}
print grunnstoffvekt["H"]
```

Dictionaries er som lister foranderlige. Vi legger inn en ny verdi med en gitt nøkkel i dictionarien over, ved å skrive for eksempel grunnstoffvekt["Be"] = 9.012182. Vi sletter en verdi i dictionarien over ved å skrive for eksempel: del grunnstoffvekt["H"] Vi viser under begge operasjoner:

```
grunstoffvekt["Be"] = 9.012182
del grunnstoffvekt["H"]
```

1.3.9 None

None er et "ikke-objekt". Ikke tenk så mye på dette, men du kan komme til å møte på det senere, og vi vil derfor nevne det her.

```
verdi = None
```

1.3.10 Konvertering mellom typer: Casting

Det hender vi ønsker å endre en variabeltype til en annen. Dette er mulig ved hjelp av noe vi kaller casting. Vi skriver da

`variabel = ny_type(annen_variabel)`. For det meste skal vi bruke dette til å konvertere mellom heltall og flyttall, og noen ganger mellom disse og komplekse tall.

Casting er særlig aktuelt ved deleoperasjoner mellom heltall. Dette kalles heltallsdivisjon. Dersom vi deler heltallet 2 på heltallet 3, vil vi i python få svaret 0. Dette skyldes at vi ved heltallsdivisjon, bare finner det hele tallet, og kaster den såkalte resten.

For å få et svar forskjellig fra null må vi bruke flyttallsdivisjon. Dette kan vi gjøre ved å velge tallene til flyttall i utgangspunktet, ved å skrive 3.0

istedenfor 3 eller 2.0 istedenfor 2. (Dersom et tall i en divisjon er et flyttall gjøres flyttallsdivisjon.) Et annet alternativ er å caste. Vi viser et eksempel under:

```
to = 2
tre = 3
to_over_tre = float(to)/tre
```

Ofte er vi også interessert i å få et heltallssvar til slutt, etter å ha gjort utregninger med flyttall. Da kan vi konvertere tilbake ved å caste[4] (igjen).

1.4 If-forgreininger og løkker

Det hender ofte at vi vil gjøre ulike ting avhengig av verdien av en variabel, at vi vil gjenta en operasjon et visst antall ganger, eller gjenta en mengde operasjoner inntil et kriterium er oppfylt. Alle programmeringsspråk har støtte for slike forgreninger, og løkker, faktisk er de selve basisen for all programmering. Vi vil nå gi en kort gjennomgang av både if-forgreninger og for- og while-løkker.

1.4.1 If-forgreninger

Det hender ofte at vi vil gjøre noe bare dersom et kriterium er oppfylt. Vi kan da bruke en if-forgreining til å sortere ut tilfellene der kriteriet er oppfylt. If-forgreningen starter da med ordet if etterfylt av et bolsk uttrykk. Deretter kommer et kolon, og så noen linjer med tekst som er rykket inn. Den innrykkede teksten kalles en blokk, og vi kaller den if-blokken. Det bolske uttrykket er kriteriet vårt. Hvis det er sant, gjør vi det som står inne i if-blokken.

Vi demonstrerer dette ved et eksempel. Vi bruker formel 1.1 øverst i kapittelet. Formelen kan for eksempel beskrive høyden til en ball med en viss masse og startfart. Vi vil vi skrive ut høyden til ballen etter at det har gått en viss tid. La oss tenke oss at når høyden er null er vi ved bakkenivå. Hvis høyden er lavere enn dette, vil det kanskje være unaturlig å skrive ut resultatet. Vi skriver nå en programsnutt som bare skriver ut høyden hvis den er høyere enn null. Vi bruker også variable, for lettere å se hva størrelsene i formelen er. Det gjør at programmet er enklere å forstå både for andre, og av oss selv etter at vi har skrevet programmet.

[4] Når vi gjør dette må vi huske på at casting fra flyttall til heltall bare innebærer å kaste alle tall etter komma. D.v.s. at vi runder ned til nærmeste heltall under. Hvis du vil ha nærmeste heltall må du først bruke funksjonen round. Du skriver `heltall = int(round(desimaltall))`

```
# -*- coding: utf8-*-

v0 = 5
g = 9.81
t = 0.6
y = v0*t - 0.5*g*t**2

if y >= 0:
    #Inne i if-blokken, kjøres hvis y er større eller lik 0:
    print "Med startfart %d er ballen %f meter \
        over bakken etter %f sekunder" %(v0,y,t)

"""
Dette er et program som regner ut en balls høyde
og finner ut om den har landet. Dersom ballen
ikke har landet, skriver den ut høyden ved den
gitte tiden og startfarten.
"""
```

Vi skriver altså ut høyden bare hvis den er større enn null. Alt som er inne i if-grenen må ha samme innrykk, siden innrykket definerer start og slutt på blokker i python. En god regel kan være alltid å bruke fire mellomrom for hvert innrykk, eller ett tabulatorinnrykk. (Prøv å bestem deg for en av delene, prøv å ikke blande de to i samme program.)

Legg forøvrig merke til setningen med # foran i if-blokken. Denne setningen er kommentert ut. Vi kan da skrive kommentarer som forklarer programmet. Vi bruker # dersom vi vil kommentere ut resten av det som står på en linje. Dersom vi vil ha en lengre kommentar, på flere linjer, bruker vi tre enkle anførselstegn. Alt som står etter disse vil da bli kommenteret ut. For å avslutte kommentaren bruker vi tre nye enkle anførselstegn.

Vi ser også en litt kryptisk kommentar øverst. Denne kommentaren gjør det mulig å skrive tegn som æ, ø og å i programmet, ved å spesifisere at vi bruker tegnsettet utf8 [5]. Dette er lurt når du skriver programmer på norsk, eller vil skrive norske kommentarer. I større prosjekter kan det være lurt å skrive koden på engelsk, men i dette kurset vil vi oppfordre deg til å skrive det du føler er mest naturlig. Bruk uansett bare æ, ø og å i kommentarer eller utskriftstekst.

Dersom vi i programmet over, også vil skrive ut noe hvis høyden er for lav, bruker vi noe som kalles en else-gren. Dette er en blokk som kjøres, dersom kriteriet for å gå inn i if-grenen ikke slår til. Vi viser det modifiserte programmet:

[5] Ikke alle systemer har støtte for kodesettet utf8 i python, og du må da bytte utf8 med latin-1.

```
# -*- coding: utf8-*-

v0 = 5
g = 9.81
t = 0.6
y = v0*t - 0.5*g*t**2

if y >= 0:
    #Inne i if-blokken, kjøres hvis y er større eller lik 0:
    print "Med startfart %d er ballen %f meter \
        over bakken etter %f sekunder" %(v0,y,t)
else:
    #I else-blokken, kjøres ellers, d.v.s hvis y er mindre enn 0
    print "Etter ",t," sekunder har ballen allerede landet"
    #Over har vi formatert teksten v.h.a tekstkonkantingering

print "Program slutt"
# Denne setningen er utenfor if-forgreiningen, og kjøres uansett
'''
Dette er et program som regner ut en balls høyde
og finner ut om den har landet. Dersom ballen
ikke har landet, skriver den ut høyden ved den
gitte tiden og startfarten. Ellers forteller
programmet oss at ballen allerede har landet.

'''
```

Vi ser her hvordan else-grenen fungerer. Vi ser også eksempel på en kommentar over flere linjer, og på formatering v.h.a tekstkonkantingering (plusstegn mellom hver bit som skal bindes sammen).

Videre kan det være at vi også vil ha en spesiell håndtering dersom vi finner tidspunktet da ballen akkurat er landet. Vi bruker da elif-forgreining. elif er en forkortelse for else-if. Det vil si at hvis ikke den første if-testen slår til, vil vi teste på ny for noe annet. Vi viser under, hvordan dette kan gjøres:

```
# -*- coding: utf8-*-

v0 = 5
g = 9.81
t = 0.6
y = v0*t -0.5*g*t**2

if y > 0:
    #Inne i if-blokken, kjøres hvis y er større enn 0
    print "Med startfart %d er ballen %f meter \
over bakken etter %f sekunder" %(v0,y,t)
elif y == 0:
    # Inne i elif-blokken, kjøres hvis y er lik 0
    print "Ballen lander etter %f sekunder"%t
else:
    #I else-blokken, kjøres ellers, d.v.s hvis y er mindre enn 0
    print "Etter ",t, " sekunder har ballen allerede landet"
    #Her har vi formatert teksten v.h.a tekstkonkantingering

print "Program slutt"
# Denne setningen er utenfor if-forgreiningen, og kjøres uansett

'''
Dette er et program som regner ut en balls høyde og
finner ut om den har landet. Dersom ballen ikke har landet,
skriver den ut høyden ved den gitte tiden og startfarten.
Dersom ballen lander akkurat på det tidspunktet vi velger,
skrives landingstidspunktet ut.
Ellers forteller programmet oss at ballen allerede har landet.

'''
```

Her er elif-blokken demonstrert. Merk forøvrig at når det bare er en verdi som skal puttes inn i tekststrengen, trenger vi ikke å bruke et tuppel med et element for å angi hva vi vil putte inn. Det holder da å angi størrelsen vi vil putte inn i tekststrengen.

1.4.2 while-løkker

Anta i eksempelet over at vi vil finne ut med et tidels sekunds nøyaktighet når ballen lander. Vi vil da begynne fra 0.1 sekunder fra ballen blir kastet opp, regne høyden, sjekke om den er større enn null, og fortsette med verdien 0.1 sekund høyere hvis den er det. Ellers vil vi skrive ut tiden vi er kommet til. Dette kan vi gjøre v.h.a en while-løkke.

En while-løkke starter med ordet while, og deretter som i en if-gren, et bolsk utsagn. Men i while-løkken går vi opp igjen og sjekker om testen har slått til på nytt og på nytt hver gang vi har kjørt while-blokken innenfor. Først når kriteriet ikke lenger er gyldig, avslutter vi løkken. Dette gjør også at vi kan skape en form for evighetsmaskin, dersom vi ikke er forsiktige. Dersom vi glemmer å oppdatere verdien på antall sekunder som har gått, for eksempel, vil vi bare teste for samme verdi igjen og igjen. Hvis ballen da er over bakken den første gangen vil den fortsette å være det igjen og igjen. [6]

[6] Dersom du kjører et program med en while-løkke, og programmet "aldri" blir ferdig,

Vi viser nå eksempelet:

```
# -*- coding: utf8-*-
v0 = 5
g = 9.81
t = 0.0

while v0*t - 0.5*g*t**2 >= 0:
   #Inne i while-løkka
    t += 0.1
''' += betyr at t blir satt til verdien t hadde før,
    pluss verdien på høyre side, altså t øker med 0.1
'''


print "Med en startfart på %d meter per sekund,\
       har ballen truffet bakken etter %f sekunder"%(v0,t)

'''
Dette er et program som finner ballens høyde hvert
0.1-te sekund og sjekker om den er større enn null
v.h.a. en while-løkke.
Når ballen har truffet bakken, slutter simuleringen,
og programmet skriver ut hvilken tid det har kommet
til.
'''
```

Vi kan også tenke oss at vi er interessert i å beholde informasjonen om høydene ved hver "måling". Vi kan da lage en liste som vi putter elementer inn i etterhvert som de dukker opp. Vi kan også lage en liste over tidene vi har målt ved. Et eksempel på hvordan det modifiserte programemt da kan se ut blir:

kan det godt være det er noe slikt som har skjedd. For å stoppe et program i kjøringen kan du trykke Ctrl D. Da vil programemt stoppe. For å teste om du har fått en evig løkkke, kan du for eksempel lage en utskrift hver gang programmet går igjennom while-blokken. Hvis denne utskriften kommer igjen og igjen i det uendelige når du kjører programmet, har du sannsynligvis en evig løkke, og du må se nærmere på sannhetskriteriet i while-løkken. Blir det noensinne usant?

```
v0 = 5
g = 9.81
t = 0.0
tider = [t] #Vi starter med lister med startverdiene
hoyder = [0]
i = 0

'''i er en tellevariabel som teller antall ganger
   vi har gått igjennom løkka.
   Den teller også indeksen på listene
'''

while hoyder[i] >= 0:
   #Inne i while-løkka
   i += 1 # Vi øker telle-variablen
   tider.append(i*0.1)
   t = tider[i]
   hoyder.append(v0*t -0.5*g*t**2)

print tider
print hoyder
'''
Dette er et program som simulerer ballens høyde,
og lagrer tiden og høyden i lister
'''
```

1.4.3 for-løkker

Noen ganger vil vi gjøre noe et bestemt antall ganger, eller for hvert element i en liste. Dette er det fullt mulig å gjøre ved hjelp av en while-løkke. Vi innfører da en tellevariabel, og teller opp hvor mange ganger vi skal gjøre noe. Når tellevariabelen er blitt større enn dette antallet (eller lik det dersom vi starter tellingen på 0), avslutter vi while-løkka. Det bolske utsagnet blir da om tellevariablen er større (eller lik) et visst tall. Dersom det er elementene i en liste vi vil gjøre noe med, eller itererere over, gjør vi dette ved å hente ut lengde av listen og så telle over elementene. Dersom vi for eksempel vil få til en pen utskrift av listene fra forrige eksempel kan vi gjøre dette på denne måten:

```
j = 0
n = len(tider)
print "|%10s|%10s|" % ("Tid", "Høyde")
print "--------------------------------------"

while j < n:
    print "|%10f|%10f|" % (tider[j], hoyder[j])
    j += 1

print "--------------------------------------"
```

Men selvom dette kan gjøres ved while, kan det gjøres enklere med det vi kaller en for-løkke.

I en for-løkke itererer vi over elementene i en eller annen liste. Dersom vi bare vil gjøre noe n ganger, gjøres dette enklest med en for-løkke over et range. En for-løkke har den fordelen, at det er mye vanskeligere å få til en evig løkke. Koden blir ofte også mer kompakt.

Til eksempelet over kunne vi iterert over en av de to listene, men siden vi må ta ut elementene fra den andre listen også, er det enklest å gjøre det ved å iterere over et range:

```
n = len(tider)
print "|%10s |%10s|" % ("Tid", "Høyde")
print "-------------------------------------"

for j in range(n):
    print "|%10f |%10f|" % (tider[j], hoyder[j])

print "-------------------------------------"
```

Vi ser at koden har blitt to linjer kortere.

Chapter 2

Dag 2: Funksjoner, pakker, brukerinput og feilmeldinger

I forrige kapittel gikk vi gjennom noen av de viktigste grunnelementene som finnes i python. I dette kapittelet skal vi lære om det siste viktige grunnelementet vi skal gå igjennom; funksjoner.

Det finnes også mange tileggspakker eller moduler til python som vi kan gjøre bruk av. Disse er en slags liste med funksjoner og typer, eller objekter, og størrelser som andre har skrevet. Disse er skrevet av flinke folk, og ofte er de også skrevet i andre språk enn python. Siden python er et relativt treigt språk, kan det å bruke slike pakker ofte være en stor fordel. Det er dessuten helt nødvendig å kunne utnytte seg av det andre har gjort fra før av, hvis man skal kunne få gjort noe særlig. Det er ikke noen vits i at alle finner opp kruttet på nytt.

I noen av disse modulene finnes det verktøy for å ta inn informasjon fra brukeren. Det er ikke bare kjekt når du skal lage programmer dom andre skal bruke, men også til programmer du lager selv. Tenk for eksempel på programmet som regner en balls høyde. Dersom du vil bruke en annen startfart, må du gå inn og endre programmet, lagre å kjøre på nytt. Dersom du isteden lar startfarten være bestemt av brukerinput kan du bestemme startfart enkelt hver gang du kjører programmet.

Til sist i dette kapittelet skal du få lære litt om feilhåndtering og om feilmeldinger. Feilhåndtering handler om å gardere seg for dumme brukere av programmet ditt, og er en viktig del av det å ta inn informasjon fra brukeren.Feilmeldinger har du sikkert allerede støtt på, å det er kjekt å vite litt mer om hva de betyr, slik at du lettere kan rette feilene dine.

2.1 Funksjoner - programmeringens oppskrifter

En funksjon i et dataprogram er ikke så veldig ulik fra funksjoner i matematikken. De er en definisjon på hvordan vi regner eller får ut noe fra noen

variable. Et eksempel på programmering av en annengradfunksjon kan være:

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

print annengradsfunksjon(4)
```

Først har vi skrevet vi def. Det er et python-ord som varsler at det kommer en funksjon. annengradsfunksjon er her navnet på funksjonen. Du kan kalle en funksjon hva du vil, men velg gjerne et navn som sier noe om hva funksjonen gjør. Etter navnet følger et tuppel med variablene som funksjonen tar inn. Denne funksjonen tar inn en variabel og gir denne variabelen navnet x. Så følger et kolon og en blokk som skal utføres hver gang vi kaller på funksjonen. De fleste funksjoner returnerer noe, men dette er ikke nødvendig. Her returnerer vi x**2. Når vi sier return går vi tilbake til der vi var da vi kalte på funksjonen, og bytter ut kallet med funksjonsverdien. Utskriften fra programmet over vil derfor være 4 opphøyd i annen, altså 16.

Selv tenker jeg ofte på funksjoner som kakeopskrifter. Når vi definerer funksjonen, er det som om vi skriver ned kakeoppskriften i kokeboka. Men det blir jo ikke kake av den grunn. På samme måte kjøres ikke innholdet i funksjonen, med mindre vi kaller på den. Variablene i tuppelet, eller parametrene til funksjonen, er som ingrediensene. I funksjonen får de navn, men de har ingen konkret verdi. Når vi kaller på funksjonen med en bestemt mengde med variable, er det som om vi sender med matvarene som oppskriften krever. Vi går da opp til funksjonen og gjør det som står der, vi baker kaken. Deretter sender vi den ferdige kaken, returverdien ned til der vi kalte på den.

Dersom vi sender med for få eller for mange argumenter, eller argumenter av feil type, vil programmet kræsje. (Det går ikke an å bake sjokoladekake hvis man prøver å starte med leverpostei og makrell i tomat.)

2.1.1 Hvorfor bruke funksjoner?

I prinsipept kunne man tenke seg at det var like enkelt bare å regne ut og gjøre det man ville i hoveddelen av programmet, uten å bruke funksjoner. Så hvorfor bruker vi funksjoner?

Det finnes mange grunner til å bruke funksjoner. Det som er lettest å forstå, er kanskje at vi vil slippe å gjøre alt fra bunnen hver gang. Det kan for eksempel være at vi trenger å regne ut $n!$ for mange forskjellig tall n, på ulike steder i et program. Man kan finne $n!$ for eksempel ved hjelp av en for-løkke, men det blir slitsomt dersom vi skal skrive denne koden igjen og igjen. Da er det fint å ha skrevet en funksjon som vi vet at gir oss $n!$ når

vi ønsker det. Når vi har skrevet den en gang, kan vi også legge den i en egenskrevet modul eller pakke som vi importerer fra. [1]

Under ser vi et eksempel på en slik funksjon, som vi tester på verdien $n = 4$. Legg merke til at vi må bruke range(1,n+1) for å få i til å ta verdien fra 1 opptil n.

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

print nfakultet(4)
```

Det som i praksis er den viktigste grunnen til å bruke funksjoner, er at det øker lesbarheten til koden. Det høres kanskje litt merkelig ut at dette er det viktigste, men når du har programmert en stund, vil du erfare at den største utfordringen ikke er å finne en løsning på hvert enkelt problem. Det vanskeligste er å finne feil, og holde rede på hva koden faktisk gjør. Ettehvert kan du også komme til å skrive kode sammen med andre, og da er det viktig at de forstår hva du har skrevet, og at du forstår hva de har skrevet.

Det finnes mange eksempler på programmerere som har tatt over andres kode. Koden har fungert godt, og programmereren skal bare utvide funksjonaliteten. Etter tallrike timer i forsøk på å forstå hva den opprinnelige koden gjør, har programmereren gitt opp og skrevet koden på nytt fra bunnen. Dette kan også gjelde gammel kode programmereren har skrevet selv, og tatt opp igjen etter at det har gått en tid.

2.2 Import av pakker

En pakke, eller modul, inneholder funksjoner, konstantverdier og typer/objekter, som vi kan tenke oss å bruke. Det finnes flere forskjellig måter å importere noe fra en modul på. Vi vil vise eksempler på hvordan vi importerer fra pakken math. Deretter vil vi gå nærmere innpå noen pakker det kan bli aktuelt for deg å bruke, og hvordan du kan finne ut hva slags funksjonalitet som ligger i en pakke du ikke kjenner fra før av.

For å importere hele math pakken skriver vi import math. Deretter bruker vi funksjonen cos (cosinus) og verdien pi fra pakken:

```
import math
print math.cos(math.pi)
```

Vi ser at vi får tak i elementene i math ved å skrive math.element. Dette leses "math sitt element", altså er det funksjonen cos i pakken math vi vil ha

[1] Pakkeimport lærer du mer om i neste avsnitt, men dersom du vil lære hvordan du kan importere selvlagde funksjoner må du lese om det i pensum til INF1100

tak i. Denne importmetoden er litt mer omstendelig enn importmetodene vi skal vise nedenfor, men kan være grei hvis vi har tenkt å bruke funksjonalitet som har samme navn som noe vi har lagt selv. Å lage variable som heter det samme som funksjoner i pakker vi importerer er uansett ikke å anbefale, men det kan være at funksjonalitet i ulike moduler som har samme navn. Da kan det være greit å bruke denne importmetoden for å skille fuksjonaliteten i de ulike pakken fra hverandre. For eksempel har pakken math en dobbeltgjenger, cmath, som inneholder de samme funksjonene og verdiene, men denne gangen for komplekse tall.

For å importere bare elementene cos (cosinus) og verdien pi fra modulen:

```
from math cos, pi
print cos(pi)
```

Vi kan nå bruke cos og pi, ved å kalle dem bare det. Da blir koden litt enklere. Det er også raskere å importere bare det man trenger fra en pakke. I de fleste programmer går alt så fort at dette ikke blir merkbart, men noen ganger trenger man den ekstra farten og den ekstra plassen i minnet, som man vinner ved ikke å importere det man ikke trenger.

Dersom man ved metoden over har problemer med navnekollisjoner kan man løse det på følgende måte:

```
from math import sin as math_sin, pi as math_pi
print math_sin(math_pi)
```

Vi importerer funksjonaliteten men gir den samtidig et annet navn.

Som regel har vi tid til å importere hele pakken. Det kan også være at vi etterhvert vil utvide programmet slik at vi bruker mer av funksjonaliteten i pakken. Da er den greieste måte å importere som følger

```
from math import *
print cos(pi)
print sin(e)
#Her regner vi også ut ut sinus til e,
#e og sin er importert fra math.
```

* er her en generisk måte å si "hva-som-helst" eller "alt" på. Vi vil benytte oss av * også senere, og denne bruken av * er den samme også i kommandolinjen i linux.

2.2.1 Noen pakker, og hvordan du finner ut mer om dem

Ovenfor har du sett bruk av pakken math. I dette kurset vil vi også bruke pakkene sys, numpy, scitools.all og muligens også time og random. time, random, sys og math er standardpakker i python. For å lese mer om disse og andre standardpakker i python går du inn på http://docs.python.org/modindex.html.

Her ligger en liste over alle standardmodulene sortert alfabetisk. Du finner fram til pakken du vil se på og klikker deg inn på den. Du får da fram en side der det står om funksjonene og konstantverdiene du kan importere fra pakken. Vi vil prøve dette i oppgavene.

Pakkene scitools.all og numpy er ikke standard. Vi vil gå igjennom eksempler på det meste av det vi kommer til å bruke fra disse pakkene her eller i oppgavene, ellers referes du til læreboka til INF1100 som kan lastes ned fra følgende lenke:

http://www.uio.no/studier/emner/matnat/ifi/INF1100/h07/pensumliste.xml.

numpy innholder alt i math og cmath med en enhetlig håndtering for komplekse og reelle tall, og en god del funksjonalitet for arrayer. arrayer likner lister, men er raskere, og har endel ekstra-funksjoner. For eksempel kan du få noe som likner et range, men med flyttall istedenfor heltall. Dette kalles et linspace.

2.3 Brukerinput - sys og raw_input

Ofte ønsker vi at programmene skal ta input under kjøring. For eksempel kan vi ha et program, som regner ut temperaturer i Farenheit fra temperaturer oppgitt i Celsius, eller et program som regner ut forskjellige ting fra måleverdier. Da vil vi kunne gi disse verdiene ved kjøring, istedenfor å måtte skrive om programmet hver gang.

2.3.1 raw_input

Den enkleste måten å få input fra brukeren på er ved å bruke raw_input-funkjsonen. Denne ligger som standard i python, og du trenger ikke importere noen pakker for å bruke den. raw_input-funksjonen bruker du ved at du setter variablen som brukeren skal oppgi lik resultatet fra raw_input-funksjonen. raw_input-funksjonen tar et tekststrengparameter. Denne tekststrengen kommer ut i kommandovinduet som et spørsmål til brukeren når vi kjører programmet. Etter at spørsmålet er skrevet ut, venter programmet til brukeren skriver

Hvis vi for eksempel vil ha antall grader i Celsius, her kalt C, fra brukeren blir koden:

```
s = raw_input('Oppgi temperturen i grader celsius')
C = float(s)
```

Grunnen til at vi først bruker en variabel s, og så caster verdien til en float, er at resultatet fra raw_input-funksjonen alltid er en tekststreng. Jeg vil anbefale at du bruker casting når du skal motta verdier fra bruker. Du trenger ikke lage en hjelpevariabel, men kan godt caste direkte. Tenk også over hva slags type det er naturlig å få. Dersom du for eksempel spør om

hvor mange ledd av en sum man skal regne ut, er det naturlig å forvente et heltall.

Det finnes også en konverteringsmetode som konverter alt til "dets egentlige type". Da konverterer du v.h.a. funksjonen eval. eval evaluerer tekststrengen den får som parameter til det den ville vært hvis den var normal python-kode. Den kan brukes for det meste, men feilretting, feilhåndtering o.l. blir lettere dersom du caster direkte til riktig type. Bruk derfor helst kun eval når det er nødvendig, d.v.s. når du ønsker at bruker skal kunne oppgi en python-formel e.l.. Et eksempel på denne bruksmåten kan være et enkelt pythonkalkulatorprogram:

```
#-*- coding: utf8 -*-
from math import *
print 'Velkommen til pythonkalkulatoren \n'

while s != 'STOPP':
    print 'Skriv inn uttrykket du ønsker å evaluere'
    s = raw_input('For å avslutte skriv STOPP. \n')
    if s != 'STOPP':
        print '\n' + "Resultatet ble: " + eval(s)

print 'På gjensyn'
```

l

Tekststrengen $\backslash n$ betyr ny linje.

Funksjonen exec har de samme mulighetene, men denne kan også evaluere vilkårlige python-uttrykk. Vi skal ikke gå nærmere innpå denne, men bare avslutte dette avsnittet med å vise et eksempel på hvordan vi kan definere funksjoner fra brukeren v.h.a. denne:

```
from math import *
formel = raw_input('Skriv en formel med variabelen x: ')
kode = """
def f(x):
    return %s
"""% formel
exec(kode)
print f(4)
```

Dersom funksjonen vi oppgir er sin(x)*cos(3*x) + x**2, vil kodebiten over være identisk med det følgende:

```
def f(x):
    sin(x)*cos(3*x) + x**2
print f(4)
```

2.3.2 Kommandolinjeargumenter og pakken sys

Ved å importere pakken sys, kan vi også ta inn argumenter som gis idet man kaller på programmet (ved `python program.py muligens_mer_tekst`.

Med pakken sys følger nemlig en variabel som heter argv. sys.argv er en liste med hvert ord som følger etter "python". Altså er navnet på programmet det 0-te elementet i denne lista. Hvert ord eller element i lista er som fra raw_input, lagret som tekststrenger. Vi får tak i dem ved hjelp av indeksering i lista, og konverterer dem på samme måte som for raw_input

Begynnelsen på programmet for å konvertere celsius til Farenheit ovenfra, skriver vi om til ta inn temperaturen i celsius fra kommandolinja på føgende måte:

```
import sys
C = float(sys.argv[1])
```

Vi regner da med at temperaturen i celsius blir skrevet etter programnavnet. Vi kan ta inn et vilkårlig antall kommandolinjeargumenter. Ellers er bruken av kommandolinjeargumenter ikke veldig ulik bruken av raw_input.

Det går også an å ta inn kommandolinjeargumenter som opsjoner (eng. options) v.h.a pakken getopt. Vi skal ikke gå nærmere innpå denne her, men referer leseren til læreboka i INF1100, og pythons modulindeks for videre lesning om dette.

2.3.3 Feilhåndtering

Når man tar input fra brukeren vil det lett oppstå feil og problemer i programmet. For eksempel kan brukeren unnlate å oppgi kommandolinjeargumenter, oppgi argument av feil type (tekststreng istedet for tall), etc. Dersom vi ikke bruker feilhåndtering for slike situasjoner vil programmet lett kræsje. I neste avsnitt skal vi lære om feil og feilmeldinger. På slutten av dette avsnittet forklarer vi hvordan du kan håndtere feil i forbindelse med brukerinput.

2.4 Feil og feilmeldinger

Når vi skiver programmer oppstår det alltid feil. Dette gjelder uansett om man er nybegynner, eller en meget erfaren programmerer. Når feilen du har gjort, er slik at programmet ikke kjører, får du en feilmelding. I feilmeldingen får du beskjed om hvilken linje feilen er på, og hva slags feil det dreier seg om. Feilmelding gir som oftest en relativt god beskrivelse av hva som er galt, så ikke få panikk. Les feilmeldingen og sjekk den aktuelle linjen.

Ofte får du også gjentatt linje det er noe galt med i feilmeldingen, og det er er gjerne laget en liten pil, som viser omtrent hvor på linja feilen er. Du får også vite hva slags feiltype det er snakk om, og som regel følger det med en liten setning, som utdyper hva feil gikk ut på.

Under står det et lite avsnitt om hver av de vanligste feiltypene som kan oppstå i et python-program, og litt om hva som sannsynligvis har gått galt, når du får denne feilmeldingen. Deretter forklarer vi kort hvordan du håndterer mulige feil ved input fra bruker.

Vi anbefaler deg å lage et eseløre på disse siden, slik at du lett kan slå opp, når du programmerer og finner en feil. Vi oppfordrer deg også til å gå inn på http://folk.uio.no/sindrf/python/

2.4.1 SyntaxError

Her har du feil syntaks. Det vil si at du mangler et tegn som kolon, en linje har feil innrykk e.l. Når det gjelder denne feiltypen kan det hende at linjen med feilen, ikke er den linjen du får oppgitt i feilmeldingen. Sjekk da linjen over, eller den forrige linjen med tekst, det er vanligvis der feilen er. Feil innrykk kan også gi feiltypen IndentationError[2]. IndentationError er en type syntaksfeil, men er litt mer spesifikk.

Hver også oppmerksom på syntaksfeil som skyldes at du ikke har lukket en parentes. Dersom du har glemt å lukke en parentes, vil kompilatoren lete etter slutten på parentesen helt til den kommer til slutten av programmet. Det betyr at du får beskjed om at feilen skjedde på siste linje i programmet, selvom den egentlig oppstå langt høyere opp i koden.

2.4.2 NameError

Denne feilen oppstår når du prøver å få fatt i en variabel du ikke har initialisert, eller å bruke en metode som ikke er definert. Vanligvis skylles dette stavefeil i navnet ett eller annet sted. Den kan også skyleds at du har glemt å importere pakken der du skal hente variablen eller funksjonen fra. Feilmelidingen du får er som oftest ganske god, men du må antagelig sjekke litt lenger opp i programmet for å finne ut hva som er feil.

2.4.3 IndexError

Denne feilen oppstrå når du prøver å få tak i et element utenfor en liste. Dette skjer som oftest dersom du prøver å få kommandolinjeargumenter som ikke er oppgitt (mer om dette i neste kapittel), eller hvis du iterer en liste litt for lenge.

2.4.4 TypeError

Du sender et argument av feil type til en funksjon. For eksempel importerer du math, og prøver å finne sinus til `"Hei"`. Det kan også skje hvis du glemmer å caste tekst du har fått fra brukeren.

[2]Indentation betyr innrykk på engelsk

2.4.5 Value

Denne feilen oppstår for eksempel når du bruker en variabel du bruker har riktig type, men feil verdi. Dette kan skje hvis du prøver å finne logaritmen til et negativt tall, eller hvis du prøver å caste tekststrengen "Hei" til et heltall. Den første typen feil, unngår vi, ved å lese om funksjonene vi bruker, og holde tunga rett i munnen når vi sender med argumenter. Den andre typen feil oppstår som regel når brukeren skriver in feil eller ulovelig input. Dette må vi håndetere med feilhåndtering.

2.4.6 Feilhåndtering ved brukerinput

Når brukeren gjør sitt inntog i programmet ditt, kan det lett oppstå feil. Særlig dersom brukeren ikke er identisk med programmereren, eller det er lenge siden programmet ble skrevet, må man regne med at feil vil oppstå. Et godt program håndterer slike "brukergenererte" feil, gir en "brukervennlig" feilmelding, og sender brukeren til en ny runde, eller kaster brukeren ut av programmet.

Feilhåndtering gjør vi ved hjelp av konstruksjonen try-except:

```
try:
    <utsagn>

except:
    <utsagn>
```

Det som skjer er at datamaskinen prøver å kjøre alle kodelinjene som står inne i try-blokken. Dersom en av disse går feil, d.v.s at programmet i prinsippet skal kræsje her, hopper datamaskinen videre til except-blokken. Dersom brukeren ikke blir kastet ut her, vil resten av programmet bli kjørt.

Som eksempel viser vi en mulig feilhåndtering for programmet, der vi ber brukeren om en temperatur i celsius med raw_input. I dette programmet vil vi gi brukeren en ny sjanse, dersom vedkommende ikke har gjort det riktig (skrevet en tekst istedenfor et tall e.l.). Vi lager da en løkke som gjentaes helt til det blir riktig:

```
mangler_verdi = True

while mangler_verdi:
    try:
        C = float(raw_input('Oppgi temperturen i grader celsius'))
        mangler_verdi = False
    except:
        print "Du maa oppgi et tall!"
```

Ofte vil vi skille mellom ulike typer feil. Dersom det er kommandolinjeargumenter vi er ute etter, vil vi for eksempel gi brukeren en ulik feilmelding dersom det mangler kommandolinjeargumenter (IndexError), eller verdien er en tekst istedenfor et tall (ValueError). Vi kan da lage flere exceptgrener som mottar ulike feil. Dette kan uansett være kjekt for feilretting.

Dersom det i koden over hadde vært en syntaksfeil inne i try-blokken, ville programmet alltid ha kjørt except-grenen og kommet i en evig løkke, uansett hvor flik brukeren var til å oppgi tall. Vi viser nå hvordan versjonen med kommandolinjeargumenter, skrives om med feilhåndtering:

```
#-*-coding:utf8-*-

import sys

try:
    C = float(sys.argv[1])

except IndexError:
    raise IndexError, \
    'Du må oppgi temperatur som kommandolinjeargument!'

except ValueError:
    raise ValueError, \
    'Temperaturen må oppgis som et tall.\
"%s" er ikke et tall' % sys.argv[1]
```

Her reiser (eng: raise) vi feilen, vi får da en feilmelding som likner den ved vanlige feil, med linjeangivelse, men istedenfor den "vanlige" feilmeldingen, får vi feilmeldingen vi har lagt inn i programmet. For de fleste brukere, vil en slik feilmelding kunne være mer opplysende (dersom vi tenker oss om når vi skriver den).

Vi kan også velge å ikke bruke raise for å hente feilen. Ved bruk av kommandolinjeargumenter må vi da være nøye med å kaste ut brukeren manuelt dersom vi ikke skal få NameError senere i programmet, fordi en variabel ikke er initialisert. Det er selvsagt mulig å gjøre dette ved å plassere hele programmet inne i try-blokken. Dette er ikke å anbefale, da det blir vanskeligere å teste og feilsøke koden. Det ser også uryddig ut. Når du bruker try-except bør du bare la utsagnene du vet du kan få feil på, være inne i try-blokken. For å kaste ut brukeren, bruker vi en funksjon fra sys-modulen, funksjonen exit. Den tar inn et tall, og avslutter programmet. Alle tall har i prinsippet samme effekt, men det er vanlig å unngå tallet 0, da dette betyr "ingenting gikk galt". Her vil vi bruke 1:

```
import sys

try:
    C = float(sys.argv[1])

except IndexError:
    'Du må oppgi temperatur som kommandolinjeargument!'
    sys.exit(1)

except ValueError:
     'Temperaturen må oppgis som et tall. \
        "%s" er ikke et tall' % sys.argv[1]
     sys.exit(1)
```

Chapter 3

Dag 3: NumPy, scitools og plott

I de to foregående kapitlene har vi brukt mesteparten av tiden på grunnleggende kunnskaper og funksjoner. Vi skal nå gå videre på en av de viktigste applikasjonene, plotting og grafisk presentasjon. Vi vil her gi deg resten av verktøyet du trenger for virkelig å kunne "gjøre noe nyttig". Å kunne plotte funksjoner og data er både nyttig og tilfredstillende.

3.1 list comprehensions og numpy-arrayer, vektorer

For å kunne plotte noe trenger datamaskinen punkter, dette være seg om vi skal plotte et målt datasett, eller en kjent funksjon. Disse lagrer vi helst i lister, gjerne i par av avhengige variable (akseparametre).

Å initialisere lister til å anta funksjonsverdier, kan fort kreve mange linjer med kode, dersom vi bruker for-, eller while-løkker, en klar forenkling av dette ligger i list comprehensions, en form for implisitt for-løkke. Kort og komprimert kode er ikke bare lettere å lese igjennom, og dermed å feilsøke, den er også ofte mer effektiv, og dette er tilfellet for list comprehensions.

Ønsker vi enda større effektivitet på koden vår, må vi bruke den mer effektive arrayen. En array er begrenset til å ha en bestemt type for alle elementer, for eksempel kan alle være heltall. En array bør helst ha en kjent lengde ved initialisering, og de finnes ikke i standard python. Derimot finnes de i tilleggspakken NumPy. Vi kan vektorisere en rekke operasjoner, blant annet funksjoner, disse kan da utføres på hele arrayen under ett, og det på en meget effektiv måte. Ellers likner de svært mye på lister.

3.1.1 list comprehensions

Dersom vi vil ha en liste der hvert element er en funksjon av elementer i en annen liste, kan vi initialisere dette ved en forløkke. Anta for eksempel, at

vi vil ha en liste med kvadrattallene, for alle tall fra 1 til 20. Ved en for-løkke skriver vi det som følger:

```
kvadrattall = range(1,21)

for k in kvadrattall:
    k = k**2

print kvadrattall
```

En mye mer effektiv måte å skrive det samme på, er ved list comprehensions. Vi flytter da for-løkka fra en eksplisitt for-løkke der vi legger inn nye liste-elementer, til en implisitt for-løkke i definisjonen av listen, som følger:

```
kvadrattall2 = [k**2 for k in range(1, 21)]
print kvadrattall2
```

Vi ser at kodebiten er halvert. Vi kan også bruke list comprehensions til å lage nestede lister. Tenk deg at vi vil lage en liste med samhørende temperaturverdier i celsius og farenheit for hver femte grad celsius fra -10 til 50 grader. Formelen for temperaturer i Farenheit er:

$$F = \frac{9}{5}C + 32 \tag{3.1}$$

Vi implementerer dette ved en list comprehension som følger:

```
def C2F(C):
    return (9.0/5)*C + 32

temperaturtabell = [[C, C2F(C)] for C in range(-10, 55, 5)]
```

3.1.2 Arrayer

Mer effektivt en list comprehensions er arrayer. I begynnelsen av dette avsnittet, fortalte vi om forskjellene mellom arrayer og lister. Vi viser nå noen måter å opprette og å bruke arrayer på. Det første vi må gjøre for å kunne bruke arrayer, er å importere numpy-modulen:

```
from numpy import *
```

En enkel måte å få en array på, er å konvertere en liste til en array. Dersom listen, her kalt r, tilfredstiller kravene til en array, d.v.s. at alle elementene er av samme type, gjøres dette ganske enkelt:

```
a = array(r)
```

Dersom vi ikke har bestemt oss for innholdet i arrayen (vi vil kanskje sette det i en for-løkke e.l.), kan vi starte med en array av nuller. Vi bruker da funksjonen zeros. Parametrene til denne er for det første antallet elementer (nuller) vi vil ha, deretter typen på elementene vi vil ha inn, (dersom vi vil ha heltall er det siste ikke nødvendig):

```
a = zeros(n, float)
```

Her lager vi en array med n nuller, og flyttall. I de fleste tilfeller er det noe slikt vi ønsker oss.

Videre kan vi som med lister få tall med jevn avstand, men i motesetning til med range, kan vi her få flyttallsverdier. Her velger vi heller ikke skrittstørrelsen, men antallet punkter. Dette er en stor fordel når vi skal plotte. Vi kan da justere punktmengden etter hvordan funksjonen ser ut. For en lineær eller tilnærmet lineær funksjon trenger vi få punkter, for en sterkt oscillerende funksjon trenger vi svært mange punkter. Vi kan altså lett justere punktmengden etter å ha sett på grafen vi får ut:

```
a = linspace(p, q, n)
```

Koden over gir oss n punkter mellom p og q (endepunktene p og q er med).

Vi får tak i enkeltelementer i et array, på nøyaktig samme måte som med lister, også biter av vektoren kan aksesseres på samme måte.

Vi kan iterere over arrayer på samme måte som over lister v.h.a. forløkker. Vi får også med en raskere måte å iterere over en bestemt tallmengde av heltall (som range). Denne funksjonen kalles xrange, og er kjappere når løkkene går mange ganger. Vi kan ikke bruke xrange til å generere arrayer på samme måte som range for lister, den kan bare brukes til å få opp farten på en for-løkke.

Funksjoner uten if-forgreninger kjøres lett på en hel array under ett. Dersom du har en array kan du da bare skrive:

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

a = linspace(1, 10, 100)
farray = f(a)
```

Dersom vi derimot har en funksjon som er definert forskjellig på ulike intervaller, må vi vektorisere funksjonen. Det gjør vi som følger (eksempelet viser Heaviside-funksjonen, en viktig matematisk step-funksjon):

```
def H(x):
    if x < 0:
        return 0
    else:
        return 1

a = linspace(-5, 5, 11)
Hvec = vectorize(H)
harray = Hvec(a)
```

3.2 scitools og plotting

Alt vi trenger for å plotte nå er selve plottepakken. Plottefunksjonaliteten kobler oss opp mot et plotteprogram på maskinen og generer plottene med det. Denne funksjonaliteten, pluss alt vi har i numpy og mye mer, finner vi i pakken scitools.all. Det vil si, at for de fleste programmer er dette alt du trenger å importere.

Plottefunksjonen kan ta masser av input, eller nesten ingenting. Syntaks er svært lik syntaksen for plotting i Matlab, og dette er ikke tilfeldig. Det enkleste eksempelet der vi bare plotter en tredjegradfunksjon viser vi under. Vi gir så et eksempel der vi bruker flere opsjoner, og prøver å forklare hva alle gjør. Vi kan ikke fortelle deg alt hva du kan gjøre her, men håper å komme igjennom det viktigste. Det vi ikke rekker her håper vi du kan finne fram til selv, i læreboka til INF1100, på internett, eller ved å spørre medstudenter og lærere når behovet melder seg.

3.2.1 Enkel plotting, plotting med mange opsjoner, flere grafer i samme plott

Først et meget enkelt eksempel. Vi oppgir bare hva vi vil ha på x-aksen og y-aksen:

```
from scitools.all import *
def f(x):
    return x**2 +x**3

a = linspace(1,10,100)
farray = f(a)
plot(a, farray)
```

For å få et plott fort, bare for å få inntrykk av en funksjon vi studerer, er noe på denne formen midt i blinken. a er array med verdier på x-aksen, farrayer ligger på y-aksen.

Til plotte-funksjonen kan vi både gi flere oppsjoner, eller vi kan sette ulike verdier etter å ha laget plottet. Tenk deg at vi ved funksjonen over ønsker å ha tekst på x- og y-aksen, en tittel og en liten avmerkning på grafen som angir hvilken funksjon vi har tegnet inn (særlig viktig når vi tegner mange grafer i samme plott, vi får da markert hvilken graf som hører til hvilken funksjon). I tillegg vil vi kunne bestemme akselengdene og lagre en kopifil på datamskinen. Etter koden over skriver vi da:

```
xlabel('a-verdier') # tekst langs x-aksen
ylabel('f-verdier') # tekst langs y-aksen

legend('a**2 + a**3') # avmerking til grafen
title('Et pent lite plott') # tittel
axis([0, 11, 0, 1100]) # akseangivelse,
#x-aksen går nå fra 0 til 10,
#y-aksen går fra 0 til 1100

hardcopy('plott.eps') # gir PostScript-fil
hardcopy('plott.png') # gir PNG-fil
```

Det samme kan vi få til ved å gi flere opsjoner til plot-funksjonen. Et tilsvarende plot-utsagn blir:

```
plot(a, farray,
        xlabel='a-verdier',
        ylabel='f-verdier',
        legend='a**2 + a**3',
        title='Et pent lite plott',
        axis=[0,11,0,1100],
        hardcopy='plott.eps',
        show=True)
```

Her har vi droppet PNG-filen. show=True sørger for at vi viser grafen, dette er default, så show-argumentet er mer interessant når vi ikke vil se plottet. Dersom vi skal plotte mange funksjoner etter hverandre, f.eks hvis vi vil lage en film, kan det være greit å sette denne til False, slikt at vi slipper å bruke tid på å få opp alle plottene.

Vi kan også bestemme hvilke farger, og hva slags type linjer vi vil ha ved å oppgi dette som parameter til plot. Dette oppgir vi direkte etter det vi skal ha på x og y-aksen. Fargen oppgis ved en bokstav, og formen på linja ved et tegn. Dersom vi vil ha blå prikker i hvert arraypunkt i funksjonen ovenfor skriver vi:

```
farray = f(a)
plot(a, farray, 'bo')
```

De ulike fargekodene er:

- gul: `'y'`
- magenta: `'m'`
- cyan: `'c'`
- rød: `'r'`
- grønn: `'g'`
- blå: `'b'`
- hvit: `'w'`

- sort: `'k'`

De ulike linjetypen er:

- sammenhengende linje: `'-'`
- stiplet linje: `'--'`
- prikket linje: `':'`
- strek-prikk: `'-.'`

I tillegg kommer mange måter å markere hvert oppgitt punkt på:

- plusstegn: `'+'`
- sirkel: `'o'`
- asterisk (stjerne): `'*'`
- punkt: `'.'`
- kryss: `'x'`
- firkant: `'s'`
- diamant (ruter): `'d'`
- trekant med spissen opp: `'^'`
- trekant med spissen ned: `'v'`
- trekant med spissen mot høyre: `'>'`
- trekant med spissen mot venstre: `'<'`
- femtakket stjerne (pentagram): `'p'`
- sekstakket star (hexagram): `'h'`
- ingen markering (standard): `None`

Alle disse mulighetene og flere til kan du få tak lese om i en interaktiv økt. Du kan da skrive `help(plot)` eller be om `pydoc` med `scitools.easyviz.plot`. En tutorial skrevet for INF1100, får du ved å skrive: `pydoc scitools.easyviz`.

3.2.2 Flere funksjoner i samme plott

For å plotte flere funksjoner i samme plott, er det igjen flere måter å gjøre det på. Vi kan for det første plotte på nytt med utsagnet hold('on') imellom. Vi viser her et eksempel med funksjonene f1(t)=t**2*exp(-t**2) og f2(t)=t**4*exp(-t**2):

```
def f1(t):
    return t**2*exp(- t**2)

def f2(t):
    return t**2*f1(t)

t = linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)

plot(t, y1)
hold('on')
plot(t, y2)
xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)', 't^4*exp(-t^2)')
title('Plott med to funksjoner i samme plott')
hardcopy('plott2.eps')
```

Den andre måten å gjøre det på, er å oppgi flere funksjoner direkte til plotte-funksjonen. Vi gjør det med de samme funksjonene og arrayene som over slik:

```
plot(t, y1, t, y2, xlabel='t', ylabel='y',
     legend='t^2*exp(-t^2)', 't^4*exp(-t^2)',
     title='Plott med to funksjoner i samme plott',
     hardcopy='plott2.eps')
```

3.2.3 Å lage en film

Det hender at vi ønsker å lage filmer av funksjoner. Det kan for eksempel hende at vi har en funksjon som utvikler seg over tid, f(x,t). Vi kan lage et tredimensjonalt plott av dette, men det vil være mer naturlig å lage en film.

En film er bare en sekvens av bilder vist etterhverandre i en ikke altfor lav hastighet. Det er dette vi skal bruke når vi lager filmer av funksjoner. Vi kan tenke oss at hvert enkelt plott er et stillbilde, tatt med en form for digitalkamera. For å få en film, må vi ta mange slike bilder, hvert ved et nytt tidspunkt i funksjonenens tidsutvikling. Men det holder ikke å ta bildene, vi må også lagre dem. Når vi har lagret alle bildene må vi kunne sortere dem i riktig rekkefølge, og så vise dem i et greit tempo. For å kunne sortere dem riktig, må de ha ulike navn det er lett å sortere. Vi gir derfor bildene i slike filmer navn som er svært like, men med en eller annen sorteringsvariabel. Det naturlige er å velge et tall som sier hvilket nummer i bildeserien vi er kommet til. Siden datamaskinen sorterer siffer for siffer, må vi ha et antall 0-er foran.

Vi viser nå et eksempel med den gausisske klokkefunksjonen:

$$f(x; m, s) = \frac{1}{\sqrt{2\pi}} \frac{1}{s} exp[-\frac{1}{2}(\frac{x-m}{s})^2] \tag{3.2}$$

Der m er midtpunktet, og s er funksjonsvidden. Vi vil se hvordan denne funksjonen (f(x)) varierer med vidden s. Vi må da lage en løkke over ulike vidder fra s = 2 til s = 0.2, der vi lager plott av funksjonen. Til slutt lager vi en film:

Koden blir:

```
from scitools.all import *

def f(x, m, s):
    return (1.0/(sqrt(2*pi)*s))*exp(- 0.5*((x - m)/s)**2)

m = 0; s_start = 2; s_stopp = 0.2
s_verdier = linspace(s_start, s_stop, 30)
x = linspace(m-3*s_start, m+ 3*s_start, 1000)
max_f = f(m, m, s_stop) # finner toppunktet på den hoyeste grafen,
#f har maks på midten og oeker naar s minker

teller = 0
for s in s_values:
    #loekke som lager hvert plott
    y = f(x, m, s)
    plot(x, y, axis=[x[0], x[-1], -0.1, max_f],
         xlabel='x', ylabel='f', legend ='s=%4.2f' % s,
         hardcopy='plott_%04d.eps' % teller)
         #Vi gir bildene navn: plott0000.eps, plott0001.eps osv
    teller += 1

movie('plott_*.eps')
# lager film av alle filer som heter plott_<et_eller_annet>.eps
```

Vi har nå laget en film, og den ligger i katalogen der vi kjørte programmet fra. Den heter som standard movie.gif. Dersom vi vil at den skal hete noe annet eller ha et annet format, må vi oppgi dette som argumenter når vi kaller movie. Da kan vi også velge hvor mange bilder vi vil ha per sekund (eng. frames per second, fps). For å endre format, må vi også sette en encoder-verdi. For å få en gif-film med 2 bilder per sekund og navnet Gaussfilm.gif skriver vi:

```
movie('plott_*.eps', encoder='convert',
      fps=2, output_file='Gaussfilm.gif')
```

Tilsvarende, for å få til en mpeg film med fire bilder i sekundet:

```
movie('plott_*.eps', encoder='ffmpeg',
       fps=4, output_file='Gaussfilm.mpeg')
```

Vi kan også bytte ut "ffmpeg" med "ppmtompeg". Det finnes også andre mer avanserte muligheter, blant annet med encoderen mencoder, men disse skal vi ikke gå innpå her.

Chapter 4

Dag 4: En applikasjon: tilfeldige tall

I dette kapittelet er et bonuskapittel om tilfeldige tall med en applikasjon til random walks. Random walks og dens slektninger Monte Carlo-metodene er en meget nyttige metoder som blir brukt i vitenskapelig sammenheng fra økonomi til astrofysikk og biologi. I oppgavesettet ligger en ekstraoppgave der du selv kan endre og lage et program som simulerer random walks, (eller virrevandring som det heter på norsk). Prøv deg gjerne på disse oppgavene.

I oppgavene ligger også en større prosjektoppgave om differensiallikninger. Bruk det du har lært i MAT-INF1100, og gjør denne oppgaven.

4.1 Bonus: Tilfeldige tall og random walks

I dette avsnittet vil vi gi en kort introduksjon til tilfeldige tall. Vi vil også vise et eksempel på hvordan disse kan brukes, til å simulere såkalte random walks

Dersom du har gjort oppgaven der vi tester de matematiske reglene, har du allerede møtt på datagenererte, tilfeldige tall, eller pseudotilfeldige tall, som de kalles. En datamaskin slik vi kjenner den, kan aldri lage fullstendig tilfeldige tall. Tallene vil alltid være en sekvens generert av en eller annen funksjon. Hvordan denne sekvensen starter og utvikler seg, vil være bestemt av et seed, eller frø. Dette er et tall vi bruker for å initialisere sekvensen. Dersom du starter sekvensen med det samme frøet, vil du alltid få den samme sekvensen av tall. Dette kan være en fordel når vi skal teste programmer for feil, men generelt ønsker vi ulike sekvenser ved hver kjøring.

I dette heftet vil vi bruke en tilfeldig-tall-generator, som per default setter frøet, ved hjelp av datamaskinens klokke. Frøet vi får derfra vil endre seg ustanselig, og vi vil mest sannsynlig aldri få den samme sekvensen av

tall. Vi kan også sette frøet til denne, men dette vil vi ikke gå nærmere innpå her.

En god tilfeldig-tall-generator gir rekker av tall som virker tilfeldige, og som har en viss fordeling eller distribusjon på et intervall. En distribusjon kan være den uniforme distribusjonen, der alle tall i intervallet, i prinsippet, er like sannsynlige. Et annet eksempel kan være Gauss-distribusjonen som har en klokkeform mot midten. Det finnes mange slike distribusjoner. De fleste tilfeldige tall-generatorene gir tall i en uniform distribusjon. Vi kan selv modifisere tallene, ved å anvende en funksjon som gir den ønskede distribusjonen på de uniformt distribuerte tallene. I dette kurset vil vi bare se på, og bruke uniformt distribuerte tilfeldige tall.

I oppgavene der vi testet de matematiske reglene, fikk vi de pseudo-tilfeldige tallene a og b i en uniform distribusjon av tall i intervallet $[A, B]$ ved følgende programlinjer:

```
import random
a = random.uniform(A, B)
b = random.uniform(A, B)
```

Vi brukte da standardpakken random. Det finnes også en egen vektorisert random-pakke i numpy. Vi vil ikke gå nærmere innpå denne i dette kurset. Dersom vi bare vil ha tilfeldige tall i det halvåpne intervallet $[0, 1)$ kan vi skrive:

```
import random
a = random.random()
b = random.random()
```

4.1.1 Bruksområder

Vi kan bruke tilfeldige tall til å simulere virkeligheten. Som regel vet vi ikke alt om et system. Når vi kaster terning, for eksempel, vet vi ikke hva utfallet vil bli. Derimot vet vi noe om hvilke utfall vi kan få, og med hvilke sannsynligheter. I tilfellet med terningene, vet vi at utfallet vil bli et av heltallene 1, 2, 3, 4, 5 eller 6, og at sannsynligheten for å få en bestemt av disse er $\frac{1}{6}$. Dette kan vi simulere ved å dele intervallet fra 0 til 1 i seks like biter, og trekke et tilfeldig tall her. Vi tester hvilken del av intervallet det tilfeldige tallet ligger i, og bestemmer fra dette hvilket utfall vi har fått

Følgende kodebit vil da simulere et terningkast:

```
import random as random_number
r = random_number.random()
if 0 <=  r < 1./6:
    r = 1
elif 1./6 <= r < 2./6:
    r = 2
elif 2./6 <= r < 3./6:
    r = 4
elif 3./6 <= r < 4./6:
    r = 5
elif 4./6 <= r < 5./6:
    r = 3
else:
    r = 6
```

Det finnes en også egen funksjon for å generere tilfeldige heltall. Med den kan vi simulerer terningkastet ved å trekke et tilfeldig heltall i intervallet $[1, 6]$. Nå finner vi utfallet direkte:

```
import random as random_number
r = random_number.randint(1, 6)
```

Det er ikke bare åpenbare tilfeller, som kortspill og terningkast, der det kan lønne seg å simulere. Endel matematiske problemer er ikke mulige å løse analytisk, og det er enklere å finne en tilnærming til svaret ved å simulere situasjonen. På mange måter likner også simuleringen med tilfeldige tall mer på det som skjer når vi gjør en måling på labben. Vi vet ikke på forhånd hva utfallet blir, men betingelsene vi gjør forsøket under, og vår kunnskap om disse, legger noen begrensninger på hva utfallet kan være.

En simulering med en løkke der vi generer mange tilfeldige tall, og hver løkkegjennomgang er som en måling, kan være en billig og effektiv måte å gjøre et forarbeide til et forsøk på, eller ta opp en relativt realistisk måleserie uten å foreta forsøket i praksis.

Innen statistisk mekanikk, kvantemekanikk og kjemi, kan simuleringer også være en svært god måte å forstå dynamikken i det som "egentlig" skjer, på. Vi vet sjelden alt om verden, og det finnes også tilfeller der det heller ikke er mulig å ha full kjennskap til systemet.

I det neste avsnittet skal vi gjennomgå en enkel modell for randomwalks, dette er i seg selv en meget viktig metode, og den viser oss også endel av prinsippene i de meget utbredte Monte Carlo-metodene.

4.1.2 Random walks

Tenk deg en partikkel, kanskje et gassmolekyl, som beveger seg i én dimensjon. La oss si at det er like sannsynlig at partikkelen beveger seg mot høyre, som at den beveger seg mot venstre. Vi vil simulere partikkelens bane, og finne ut hvor den har havnet etter 100 skritt. Anta at partikkelen starter i posisjonen $x = 0$. en bevegelse mot høyre tilsvarer en økning i

posisjon på 1. En bevegelse mot venstre tilsvarer en minkning i posisjon på 1.

Vi bruker generatoren for tilfeldige heltall og får følgende kode:

```
import random as random_number

'''
grunnen til denne importmetoden, er at modulen
random i numpy ellers vil kunne overskrive den
standariserte random-modulen, dersom vi
importerer numpy etterpå med from numpy import *
'''

ns = 100                        # antall skritt
posjon = 0     # partiklen starter ved x=0
HOYRE = 1;  VENSTRE = 2             # konstanter

for skritt in range(ns):
    forflytning = random_number.randint(1,2)  # flip coin
    if forflytning == HOYRE:
        posjon += 1   # partikkelen flytter seg mot hoyre
    elif forflytning == VENSTRE:
        posisjon -= 1   # partikkelen flytter seg mot venstre

print posisjon
```

Vanligvis vil vi ikke bare simulere en partikkel. La oss tenke oss at vi har np partikkler som befinner seg i en startposisjon. Vi lar alle partiklene virrevandre fra dette punktet, og finner ut hvor hver av dem har endt opp etter ns skritt. Denne situasjonen likner en situasjon der vi åpner veggen til en boks med gass, som tidligere har vært lukket. Hvis vi igjen ser på 100 skritt og begynner med fire partikler, får vi koden:

```
import random as random_number
import numpy
ns = 100                         # antall skritt
np = 4                           #antall partikler
posjoner = numpy.zeros(np)     # partiklen starter ved x=0
HOYRE = 1;  VENSTRE = 2             # konstanter

for skritt in range(ns):
    for p in range(np):
        forflytning = random_number.randint(1,2)  # flip coin
        if forflytning == HOYRE:
            posjon[p] += 1   # partikkelen flytter seg mot hoyre
        elif forflytning == VENSTRE:
            posisjon[p] -= 1   # partikkelen flytter seg mot venstre
```

Vi kan legge inn en form for animasjon av partiklenes bevegelse dersom vi til slutt i den ytre for-løkken plotter, og lager en liten pause for å få animasjonen til å fungere. Koden for dette følger under, men den forutsetter at vi importerer scitools.all og time ved utsagnene `from scitools.all import *` og `import time`:

```
    plot(positions, y, 'ko3', axis=[xmin, xmax, -0.2, 0.2])
    time.sleep(0.2)  # pause
```

Vi må også sette $xmax$ og $xmin$ til den maksimale og minimale posisjonen, vi kan få i hele forløpet, ns og $-ns$, i vårt tilfelle -100 og 100. Det er derimot svært liten sannsynlighet for at dette skal skje, og vi kan derfor normalt sette $xmax$ og $xmin$ til litt lavere verdier. En god tommelfingerregel er $2sqrt(ns)$ og $-2sqrt(ns)$. Vi lar partikklenes posisjon langs y-aksen være null i alle tilfellene, (vi lager en array med zeros), men vi trenger litt plass for å se partikellbanene. I en av ekstraoppgavene skal vi også se på randomwalks i to dimensjoner. Da blir grafikken på de tilsvarende grafene langt mer interessant.

I statistisk fysikk er vi normalt også interessert i å finne gjennomsnittsposisjonen til partiklene, standardavviket til posisjonene, og vi vil også gjerne kunne tegne et histogram over posisjonene. Med scitools og numpy importert kan vi bruke funksjonen `compute_histogram` fra scitools, og funksjonene `mean` and `std` fra numpy for å finne dette.

```
mean_pos  = mean(positions)
stdev_pos = std(positions)
pos, freq = compute_histogram(positions, nbins=int(xmax),
                              piecewise_constant=True)
```

Når vi ser på utviklingen av disse størrelsene vil vi oppdage at med tilstrekkelig antall partikler, vil gjennomsnittsposisjon alltid ligge omkring 0. (Det er som om partiklene til sammen knapt nok har flyttet seg.) Når vi ser på standardavviket, og utviklingen av histogrammen vil vi derimot få et annet bilde. Standardavviket vil øke ettersom vi lar skrittmengden øke, først fort, så langsommere. Histogrammet vil ta form av en topp omkring x = 0.

Vi kan plotte partiklenes vandring som før, og legge inn gjennomsnittsverdien, og det positive og negative standardavviket som vertikale linjer (det siste viser bredden på fordelingen). Vi lager de vertikale linjene ved følgende programlinjer:

```
xmean, ymean   = [mean_pos, mean_pos],     [yminv, ymaxv]
xstdv1, ystdv1 = [stdev_pos, stdev_pos],   [yminv, ymaxv]
xstdv2, ystdv2 = [-stdev_pos, -stdev_pos], [yminv, ymaxv]
```

Der `yminv` og `ymaxv` er maksimums- og minimumsverdiene for y i plottet.

For å plotte partiklene som sirkler som før, legge inn de vertikale linjene, og tegne histogrammet oppå det hele:

```
plot(positions, y, 'ko3',     # particles as circles
     pos, freq, 'r',          # histogram
     xmean, ymean, 'r2',      # mean position as thick line
     xstdv1, ystdv1, 'b2',    # +1 standard dev.
     xstdv2, ystdv2, 'b2',    # -1 standard dev.
     axis=[xmin, xmax, ymin, ymax],
     title='random walk of %d particles after %d steps' % \
           (np, step+1))
```

Vi lager plottene ved hvert skritt i virrevandringen. Vi må også tenke oss nøye om før vi velger utvidelsen på y-aksen. I det fullstendige programmet, som er hentet fra INF1100, og som du finner på nettsiden

http://vefur.simula.no/intro-programming/src/random/

under navnet `walk1Ds.py`, har de valgt maksverdien til å være 110% av den maksimale verdien på histogrammet (`max(freq)`). Samtidig har de valgt at aksene bare skifter når verdien endrer seg mer enn 0.1 fra den forrige verdien. (Hver oppmerksom på at jeg i min eksempelversjon her har oversatt mange av variabelnavnene til norsk, og at jeg har byttet ut INF1100-versjonens mynt- og kron- variabelnavn med forflytning, og høyre og venstre.)

Forøvrig kan det vises at histogrammet nærmerer seg grafen til den gaussiske klokkefunksjonen fra oppgavene. Gjennomsnittsposisjonen vil som sagt være 0 og standardavviket $\sqrt{n}$ når antall partikler øker mot uendelig.

Som eksempel på en direkte applikasjon av virrevandring, kan det nevnes at virrevandring modellerer fotoners reise ut av sola. Fotonene blir laget ved hydrogenfusjon i solkjernen. Deretter virrevandrer fotonene ut av sola. Vi kan godt forenkle problemet til å se på sola som et endimensjonalt objekt, med en utstrekning i begge retninger lik dens radius. Vi kan også modellere fotonforflytningen som skrittvis vandring i enten den ene eller den andre retningen. Vi kan også anta at når fotonet først er ute av sola, går det ikke tilbake inn igjen. Dersom vi gjør denne skrittavstanden passelig kort, og bruker lyshastigheten til å beregne hvor lang tid et slikt skritt tar, får vi en god modell for å finne ut hvor lang tid et foton i gjennomsnitt bruker på veien ut av sola.

Vedlegg: Nyttige kommandoer

I dette kapittelet har jeg lagt ved noen nyttige kommandoer til terminalvinduet. Der det er naturlig å putte inn navn på noe du lager, sletter eller flytter har jeg markert dette ved å skrive hva som skal settes der på norsk.

For å starte IPython:

```
ipython
```

For å åpne emacs

```
emacs &
```

&-tegnet er ikke nødvendig, men gjør det mulig å fortsette å bruke kommandolinjevinduet, uten å lukke emacs igjen. Du kan også velge hvilken fil du vil ha åpnet med en gang. Dette gjør du ved å skrive navnet på filen mellom emacs og &-tegnet. Det siste kan du også gjøre dersom du ikke har laget filen før, da oppretter du den. På samme måte som du starter emacs, kan du i prinsippet også starte alle andre programmer fra kommandovinduet.

Manualsider til en kommando

```
man kommandoen_du_vil_vite_mer_om
```

For å lage en ny katalog:

```
mkdir navn_på_katalog
```

For å flytte deg mellom kataloger:

```
cd
```

Hvis du bare skriver cd, havner du i toppkatalogen, skriver du .. etter kommer du en katalog opp. Ellers skriver du navnet på katalogen du vil inn i (forutsatt at den ligger inne i katalogen du er i fra før av).

For å flytte filer:

```
mv navnet_på_fila katalogen_du_vil_flytte_fila_til
```

Dersom fila ligger et annet sted enn der du er, må du skrive katalogadressen foran filnavnet. Denne kommandoen kan også brukes til å skifte navn på filer, ved å skrive det gamle navnet, og så det nye.

For å slette:

```
rm det_du_vil_slette
```

Hvis du vil slette en hel katalog, og innholdet inne må du skrive `rm -r navnet_på_katalogen`

For å liste opp filer i en bestemt katalog:

```
ls
```

Et tips; skriv "ls *.py" for å liste opp alle filer som slutter på ".py" eller liknende

Et par andre nyttige ting:

1. Når man skal hente opp en fil fra kommandovinduet, kan man fort komme til å skrive feil. Da er det en fordel at kommandolinjen har en autocomplete funksjon. Dersom du begynner en kommando eller et filnavn og så trykker tab, vil den automatisk fullføre det du vil si. Dersom det finnes flere alternativer vil den først ikke fullføre, dersom du da trykker tab igjen, vil den liste opp alternativene.

2. For å lagre i emacs kan du trykke "file", og velge "save buffer". Dette er en helt ok måte å gjøre det på, men det går betydelig raskere å lagre med hurtigtaster. Da trykker du "ctrl x-s". Hold altså inne kontrolltasten og trykk på x, slipp opp x og trykk på s.