# Introduksjon til vitenskapelig programmering
# Høst 2015
# Sandvika vgs

Jonas van den Brink
j.v.d.brink@fys.uio.no

Marie Roald
marieroa@math.uio.no

Vigdis Holta
vigdihol@uio.no

16. november 2015

# Innhold

# 1 Uke 1

Denne teksten er ment som en kort oppsummering av det vi håper å ha kommet igjen etter første uke. Programmering er en ferdighet det tar tid å lære seg, og den beste måten å lære det på er rett og slett å prøve seg frem. Vi håper derfor at dere tar dere tid til å se litt på programmering på egenhånd, og skriver noen korte, enkle, og kanskje teite programmer.

## 1.1 Installasjon

Vi kommer til å bruke programmeringsspråket *Python*. Det er i prinsippet gratis, og det finnes mange forskjellige versjoner, og mange forskjellige tillegspakker. Den enkleste måten å installere alt vi trenger for dette prosjektet er å installere en samlepakke som heter *Enthought Canopy*. Den laster du ned fra linken her

`https://www.enthought.com/downloads/`.

Når filen er ferdig lastet ned, kjører du den og følger instruksene. Du kan la alt av innstillinger stå på standard om du ikke har andre preferanser.

Når du har installert Canopy, kan du starte programmet. Siden det er en samlepakke er det en del funksjonalitet vi ikke kommer til å bruke, så ikke bli skremt av det kanskje ser litt komplisert ut.

## 1.2 Hva er egentlig programmering?

Programmering går ut på å gi instrukser til datamaskinen. Disse instruksene skriver vi inn som kommandoer i en tekstfil. Denne tekstfilen kan så *kjøres* av datamaskinen, som da tolker kommandoene vi har skrevet. Det er det som er et dataprogram. Kommandoene vi skriver må følge et bestemt programmeringsspråk, og det finnes fryktelig mange slike språk idag. Vi kommer til å holde oss til Python, et av de mest file:///usr/share/doc/HTML/en-US/index.htmlfile:///usr/share/doc/HTML/en-US/index.html

## 1.3 Vi setter igang

Om vi starter Canopy, og velger *Editor*, får vi opp et vindu der vi kan skrive et slikt dataprogram. Du skriver da koden inn i det største vinduet øverst. Du kan lagre programmet ditt med det navnet du vil, men du må legge på endelsen `.py`, for Python. Etter du har lagret programmet ditt, kan du kjøre det ved å klikke på *Run*-knappen, som er en grønn pil på toppen av editoren. En snarvei for Run er `ctrl+R`.

La oss se på et eksempel på et enkelt dataprogram:

```
# Calculating the area and volume of a football

from pylab import pi

r = 10

area = 4*pi*r**2
volume = (4./3)*pi*r**3
```

```
print "A football with radius:", r
print "Has an area of:", area
print "And volume of:", volume
```

Her er det ikke viktig at du skjønner alt som skjer i detalj, men la oss prøve å skjønne hovedtrekkene. Når vi trykker på Run-knappen, starter programmet med å tolke det som er skrevet, linje for linje. La oss derfor forklare det som skjer nedover i programmet, linje for linje.

```
# Calculating the area and volume of a football
```

Fordi den første linjen begynner med tegnet `#`, betyr det at denne linjen ikke tolkes av datamaskinen i det heletatt. Vi kaller en slik linje for en kommentar, og det er rett og slett en forklaringstekst til enten oss selv, eller andre som skal lese koden vår. Vi skjønner altså at dette enkle programmet skal regne ut arealet og volumet av en fotball. Merk også at hele linjen er en annen farge fra resten av koden, dette gjør Canopy for at det skal være lettere å tolke koden.

```
from pylab import pi
```

Denne linjen ser kanskje litt avansert ut, men her importeres rett og slett tallet $\pi$. Vi trenger $\pi$ for å regne ut arealet og volumet til en kule, men den er ikke originalt tilstede i Python, vi må *importere* den fra tillegspakken *pylab*. Det finnes veldig mange slike tileggspakker i Python, til mange forskjellige bruksområder. I vårt tilfelle inneholder Pylab alt vi kommer til å trenge.

```
r = 10
```

Her defineres det at det finnes en *variabel* som heter $r$, og at den skal være 10. Utifra kommentaren på starten av programmet skjønner vi at dette mest sannsynligvis er radiusen til fotballen. Denne linjen sier altså at radiusen er 10.

```
area = 4*pi*r**2
```

Her regnes arealet til fotballen ut fra den matematiske formelen $4\pi r^2$, og resultatet lagres i en *variabel* som kalles `area`.

```
volume = (4./3)*pi*r**3
```

Og her regnes volumet ut fra $\frac{4}{3}\pi r^3$, og resultatet lagres i `volume`.

```
print "A football with radius:", r
print "Has an area of:", area
print "And volume of:", volume
```

Til slutt kommer tre veldig like linjer. Alle bruker kommandoen `print`, som forteller Python at vi skal skrive et resultat til skjermen, slik at brukeren kan lese det. Tekst vi skriver omsluttes av fnutter: `"`, og vi ber Python skrive ut resultatene av utregningene etter teksten.

Når programmet kjøres, får vi dette resultatet:

```
A football with radius: 10
Has an area of: 1256.63706144
And volume of: 4188.79020479
```

## 1.4 Variabler og regning

I eksempelet vi nettopp så på, ble vi introdusert til et veldig viktig konsept i programmering, nemlig variabler. Variabler bruker vi når vi vil at datamaskinen skal huske på en verdi vi gir den, eller noe den regner ut.

Hvis vi for eksempel skriver

```
a = 6
```

vil datamaskinen opprette en variabel som heter `a`, som inneholder *verdien* 1. Den vil så huske på denne variabelen helt til programmet er ferdig å kjøre. Hvis vi for eksempel vil skrive ut innholdet av en variabel til skjerm, bruker vi `print` commandoen, så å skrive

```
print a
```

gjør at det skrives ut et ettall til skjermen.

Vi kan også regne med en eller flere variabler. Hvis vi for eksempel også definerer en variabel `b`, ved å skrive

```
b = 3
```

Så kan vi bruke regne med disse variablene, hvis vi foreksempel skriver

```
print a + b
print a - b
print a * b
print a / b
```

får vi følgende resultater

```
9
3
18
2
```

Merk at `a` og `b` ikke endrer seg når vi regner med dem på denne måten. Dette kan vi dobbeltsjekke ved å skrive dem ut på nytt:

```
print a
print b
```

som gir

```
6
3
```

Hvis vi ønsker å endre en variabel vi allerede har definert, kan vi redefinere den, så vi kan nå skrive

```
a = -2
```

Merk at dette *overskriver* den gamle verdien av `a`, slik at programmet ikke husker hvilken verdi den var før. Vi kan også definere en variabel ved for eksempel å skrive

```
x = -4
y = 6
z = x + y
```

Her definerer vi først `x` og `y`, og deretter `z` som blir satt til resultatet av utregningen `x+y`. Om vi nå skriver ut `z` ser vi at den blir `2`, som forventet. Merk at det som skjer her, er at høyresiden regnes ut, og resultatet lagres i `z`-variabelen. Python husker altså ikke hvor verdien 2 som lagres i `z` kommer fra, den bare husker selve verdien. Prøv for eksempel å forklare hva som vil skrives ut om vi kjører denne lille kodesnutten:

```
x = 3
y = -3
z = x + y
y = 6
print x
print y
print z
```

Vil `z` inneholde verdien 0, eller 9? Bare prøv, og se hva som skjer!

Merk at likhetstegner, `=`, har en ganske annerledes betydning i programmering enn den har i matematikk. I matematikk er vi vant til at den brukes for å angi en likhet, altså en ligning. Det har for eksempel ingenting å si om vi skriver

$$x^2 = 4 \qquad \text{eller} \qquad 4 = x^2,$$

i matte. I programmering fungerer derimot ingen av disse. Tegnet `=`, brukes i Python alltid til å definere (eller redefinere) variable. Den virker alltid ved at den regner ut det som er på høyre-side, og så lagrer resultatet i variabelen på venstre side. Det er kanskje altså mer fornuftig å tenke på det som en slags pil, som peker fra høyre mot venstre.Kodesnutten

```
x = 9
y = 4
z = x*y
```

Kan altså tolkes som

$$x \leftarrow 9$$
$$y \leftarrow 4$$
$$z \leftarrow x * y$$

Når du begynner å skjønne denne tankegangen kan vi begynne å gjøre ting som kanskje ser litt rare ut. Vi kan for eksempel skrive

```
x = 3
x = x + 10
```

Fra et matematisk ståsted ser dette fryktelig ut, ligningen

$$x = x + 10,$$

gir ingen menining. Men i programmering er dette ganske greit. Først sier vi at `x` skal være 3, så regner vi ut høyre-siden, som da vil si $10 + 3 = 13$, og så lagrer vi 13 i `x`. Du kan sjekke at det er dette som faktisk skjer, ved å printe ut `x` og sjekke selv.

Hitill har vi bare vist eksempeler på variabler som inneholder tall, men de kan også inneholde andre ting, som for eksempel tekst, sannhetsverdier, og andre, mer kompliserte ting. Vi kan også gi dem mer kompliserte navn, i praksis er dette ofte lurt, fordi det gjør det lettere for oss å huske på hva de forskjellige variablene er i et langt og rotete dataprogram. I eksempelprogrammet vi viste på starten brukte vi variabelnavn som `area` og `volume`.

La oss se på et enkelt eksempel på hvordan vi kan bruke en variabel med tekst

```
name = "Jonas"
print "Hi", name, "! Hope you have a nice day =)."
```

Merk at for å opprette en variabel med tekst, så lar vi teksten stå i fnutter: `"tekst"`, dette er for å få Python til å skjønne at den ikke skal prøve å tolke teksten som kode. En slik tekst som ikke er kode kalles en *tekststreng*. Merk også at print kommandoen vi gi er litt mer komplisert en vanlig, fordi den skriver ut tre ting. Først skriven den ut tekstrengen `"Hi!"`, merk at fnuttene ikke vises på skjermen når utskriften kommer, så skriver vi ut innholden av variabelen `name`, og så skrives den siste teksstrengen.

## 1.5 Lister

Hitil har du sett at variable har et navn, et innhold og en type. La oss se på en ny type variabel: lister. La oss si at du ikke bare ønkser at programmet ditt skal huske på et navn, men en hel skoleklasse. Det er veldig slitsomt å måtte opprette en variabel for hver enkelt elev. Det vi kan gjøre, er å opprette en enkelt variabel, hvor vi lagrer alle elevene sammen, det kan du gjøre sånn her

```
students = ["Jake", "John", "Mary", "Lucy", "Alexander"]
```

Vi ser at vi har brukt firkantparanteser: `[` og `]` for å definere en liste, og inne i listen har vi skrevet 5 navn, alle adskilt med komma. Merk også at vi definerer hvert navn som hver sin tekststreng. Når du har definert en liste på denne måten, så kan du skrive den ut med `print`, og selvfølgelig sjekke typen

```
print students
print type(students)
```

Når du gjør det, så får du følgende utskrift i terminalen

```
[' Jake ', ' John ', ' Mary ', ' Lucy ', ' Alexander ']
<type 'list'>
```

En annen ting du kan gjøre med en liste, er å sjekke hvor mange ting den inneholder, det gjør vi med `len`, som forteller deg lengden på en liste

```
print len(students)
```

forteller oss at det er 5 navn i lista.

En liste trenger ikke nødvendigvis inneholde tekststrenger, vi kan plassere hva som helst i dem. Vi kan for eksempel ha en liste med tall

```
prices = ["299", "199", "4000", "20"]
```

Eller en blanding av tall og tekst

```
my_list = ["Some text", 2, 2.3, 9, "Some more text"]
```

Du kan tilogmed legge lister inne i andre lister

```
lists_in_lists = [[0, 2, 3], ["Mary", "Lucy", "Jake"]]
```

Siden en liste kan bestå av så og si hva som helst, så pleier vi å kalle det den inneholder for elementer. En liste er en rekke elementer.

Når vi først har definert en liste, for eksempel en liste over alle elevene i en skoleklasse

```
students = ["Jake", "John", "Mary", "Lucy", "Alexander"]
```

Så kan vi gå inn i lista og hente ut ett bestemt navn. Det gjør vi med noe som heter indeksering, jeg kan for eksempel skrive

```
print students[0]
print students[3]
```

Her så betyr `students[0]` det første elementet i lista, som altså er 'Jake', mens `students[3]` betyr det fjerde navnet i lista, som er 'Lucy'. Tallet vi skriver teller altså elementer utover i lista, og vi begynner å telle på 0. Det er kanskje litt rart, men sånn fungerer det altså.

Vi kan også overskrive et bestemt element i en liste. Si for eksempel at vi har funnet ut at vi har gjort en feil, Alexander i lista over heter egentlig bare Alex! Vel, da kan vi gå inn og endre bare den delen av lista. 'Alexander' står på den 5 plassen i lista, så det er `students[4]` vi må endre, så da skriver vi

```
students[4] = "Alex"
```

Hvis vi nå skriver ut hele lista på nytt med `print students` får vi utskriften

`[' Jake ', ' John ', ' Mary ', ' Lucy ', ' Alex ']`

Så du ser at det er bare 'Alexander' som har endret seg i lista.

Du kan også legge til ekstra elementer i listen din. Si for eksempel at du har glemt en av elevene i klassen din, da kan den personen legges til som følger

```
students.append("Roger")
```

Hvis vi nå skriver ut lista får vi

`[' Jake ', ' John ', ' Mary ', ' Lucy ', ' Alex ', 'Roger']`

Merk at elementer vi legger til med `.append` havner på enden av lista.

Siden vi kan legge elementer til en allerede eksisterende liste, så kan det av og til gi mening å lage en helt tom liste. Se for eksempel på denne koden

```
my_list = []
my_list.append(1)
my_list.append(2)
my_list.append(3)
```

Her lager jeg først en helt tom liste, så begynner jeg å fylle den med tall etterpå.

## 1.6 Datatyper og heltallsdivisjon

Så langt har vi sett at variabler kan ha forskjellig typer innhold. Python har en måte å sjekke hvilken type innhold en variabel har, som vi kan bruke som følger

```
a = 4
x = 3.14
name = "Jonas"
print type(a)
print type(b)
print type(c)
```

og resultatet blir som følger

```
<type 'int'>
<type 'float'>
<type 'str'>
```

Som vil si at `a` er av type *int*, som står for integer, altså heltall, `x` er *float*, som betyr desimaltall, og `name` er *str*, altså en tekststreng.

For vårt bruk er det ikke så viktig å ha oversikt over alle disse datatypene og hvordan de oppfører seg. Men vær obs på at dere kommer nok til å gjøre litt feil med disse, så det kan være greit å tenke litt over hvordan ting fungerer i detalj i blant.

En veldig vanlig feil å gjøre for eksempel, er noe som kalles *heltallsdivisjon*. Dette er når vi har to tall som Python tenker på som heltall, og prøver å dele disse på hverandre

```
a = 5
b = 3
print a/b
```

I dette tilfellet forventer vi kanskje resultatet

$$5/3 = 1.666667,$$

men det vi får er bare 1. Det er fordi Python tenker på `a` og `b` som heltall, og tror derfor at det vi er interessert i er et heltall! For å tvinge python til å skjønne at vi vil faktisk ha desimaltall, bør vi definere `a` og `b` som desimaltall, det kan vi gjøre ved å skrive

```
a = 5.0
b = 3.0
```

eller bare

```
a = 5.
b = 3.
```

Når vi gjør en divisjon med et desimaltall og et heltall, tolker Python det som at vi vil ha "vanligdivisjon, så hvis vi er interessert i å regne ut kinetisk energi for eksempel, som matematisk sett har formelen

$$K = \frac{1}{2}mv^2,$$

kan vi bruke utrykket:

```
kinetic_energy = 1./2*m*v**2
```

Et par ting å merke seg her: Vi skriver nevneren i brøken med et punktum, for å unngå heltallsdivisjon (`1/2` hadde gitt 0), og vi skriver "opphøyd i"med `**`.

## 1.7 Matematiske funksjoner

Om vi er interessert i å regne med vanlige matematiske konstanter og funksjoner, som for eksempel $\pi$, $e^x$, $\sin(x)$, osv, ligger disse lagret i pylab pakken. Vi kan da enkelt importere dem med kommandoer på formen:

```
from pylab import pi, exp, sin
```

og da bruken dem på følgende måte:

```
print sin(2*pi)
```

Vi kommer til å vise dere hvordan dere kan lage deres egne matematiske funksjoner i Python, som for eksempel:

$$f(x) = 4x^2 + 3x + 4,$$

i løpet av de neste ukene.

Hvis vi har lyst til å importere alt som ligger i pylab pakken, kan vi skrive

```
from pylab import *
```

## 1.8 Oppgaver

### 1.8.1 Oppgave 1.1

(a) Lag et skript som skriver ut teksten "Hello, World!til skjermen.

(b) Lag et skript som skriver ut verdien av 4*5+2 til skjermen, blir resultatet annerledes om du skriver 2+4*5?

(c) Lag et skript som regner ut $2^{10}$ og skriver resultatet til skjermen.

(d) Lag et skript som regner ut $\sqrt{3}$ og skriver resultatet til skjermen. Her må du først importere `sqrt`-kommandoen (squareroot) fra pylab.

### 1.8.2 Oppgave 1.2

For å regne oss om fra en temperatur oppgitt i grader Celsius $C$, til grader oppgitt i Fahrenheit $F$, bruker vi formelen

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

(a) Lag et skript som regner ut temperaturen i Fahrenheit for en gitt temperatur i Celsius.

(b) Bruk skriptet til å finne ut hvor mange grader Fahrenheit disse tempraturene er: $20°$, $0°$, $-40°$.

### 1.8.3 Oppgave 1.3

(a) Skriv et skript der du lagrer et fornavn og et etternavn. Skriv ut en beskjed til personen.

(b) Skriv også ut antall tegn i fornavnet og etternavnet. For å gjøre dette skal du bruke kommandoen `len(name)` som gir lengden av en tekststreng, det vil si, antall karakter. For eksempel gir `len("Jonas")`, resultatet 5.

### 1.8.4 Oppgave 1.4

Ta en titt på følgende program

```
var = 10;
print var
var = var + 10*var
#var = 0
print var
```

(a) Uten å faktisk kjøre programmet, hva tror du at utskriften fra denne koden kommer til å være?

(b) Kjør koden for å se om du hadde rett.

### 1.8.5 Oppgave 1.5

Ta en titt på følgende program:

```
from numpy import *
r = 10
A = ?
O = ?
print "Arealet av sirkelen er:", A
print "Omkretsen til sirkelen er:", O
```

(a) Fiks programmet slik at A og O er hendholdsvis arealet og omkretsen til en sirkel med radius r

(b) Kjør koden for å se hva svaret ble.

### 1.8.6 Oppgave 1.6

Ta en titt på følgende program

```
x = [1]
x.append(2)
x.append(3)
print x
x[0] = 4
print x
print len(x)
print type(x)
print type(x[0])
```

(a) Uten å faktisk kjøre programmet, hva tror du at utskriften fra denne koden kommer til å være?

(b) Kjør koden for å se om du hadde rett.

# 2 Uke 2

Sist uke så vi på våre første kodesnutter og lærte å lage enkle programmer. Vi lærte hvordan vi skriver korte scripts ved å gi datamaskinen en rekke med kommandoer, og hvordan disse tolkes av maskinen når vi kjører programmet vårt. Vi så også på variable og typer, hvordan disse lages og brukes i programmering. Idag går vi videre med litt mer sammensatt programmering, vi kommer til å få bruk for alt vi har lært til nå, så vi kommer til å få god repetisjon av forrige ukes stoff på kjøpet.

## 2.1 Løkker

Når vi ønsker å gjenta biter med kode, bruker vi gjerne noe som kalles en løkke, eller *loop* på engelsk. I python har vi to typer løkken, og de er *for*-løkken, og *while*-løkken. I dag kommer vi bare til å se på for-løkker. En for løkke itererer over elementer i en liste, og utfører de samme kommandoene for hvert element.

La oss se på et enkelt eksempel

```
for name in ['John', 'Mary', 'Lucy', 'Roger']:
    print name
```

Vi ser at vi starter en for-løkke med ordet `for`, vi gir så navn til elementet, her har vi gitt det navnet `name`, og så skriver vi kommandoen `in` og spesifiserer en liste. Nå kjøres all koden med innrykk om igjen for hvert element i lista. Resultatet av denne kodesnutten blir altså

```
John
Mary
Lucy
Roger
```

Merk at vi når vi definerte for-løkka skrev lista vi iterer over rett inn, vi skal selvfølgelig også lagre listen som en variabel, og gi variabelen, altså som følger:

```
names = ['John', 'Mary', 'Lucy', 'Roger']

for name in names:
    print name
```

### 2.1.1 Range

Ofte når vi bruker løkker i programmeringssammenheng, er vi interessert i å iterere over en liste med tall. Det kan derfor være lurt å ha måter å lage store lister med tall på en enkel måte. For å gjøre dette kommer vi til å bruke Python-funksjonen `range`. Vi må fortelle range hvor vi vil at listen skal begynne, hvor den skal slutte, og hvor store steg den skal ta. Ved default vil den begynne på 0 og ta steg på 1, så de tre kommandoene

```
print range(10)      # gir stop = 10
print range(0,10)    # gir start = 0, stop = 10
print range(0,10,1) # gir start = 0, stop = 10, step = 1
```

gir det samme resultatet, nemlig

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Merk at listen sluttet på 9, og ikke 10, `range`-kommandoen gir altså en liste fra og med `start`, til (men ikke med) `stop`, med steg på `step`. La oss se på et par flere eksempler:

```
print range(1,10)
>>> [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
print range(1,10,2)
>>> [1, 3, 5, 7, 9]
```

```
print range(-10,10,5)
>>> [-10, -5, 0, 5]
```

```
print range(10, 4, -1)
[10, 9, 8, 7, 6, 5]
```

```
print range(3,30,3)
[3, 6, 9, 12, 15, 18, 21, 24, 27]
```

```
print range(100)
[0, 1, 2, 3, 4, ..., 92, 93, 94, 95, 96, 97, 98, 99]
```

Ved å bruke `range`-kommandoen på riktig vis, kan vi altså lage lister med tall på en rask måte.

### Eksempel: Matematiske summer

Som et eksempel, la oss bruke en løkke til å regne ut summen av tallene fra og med 1 til og med 1000. Det vil si:

$$S = \sum_{i=1}^{1000} i = 1 + 2 + 3 + \ldots + 998 + 999 + 1000.$$

I python, kan vi finne denne summen med følgende kodesnutt

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

print S
```

som gir svaret

$$S = \sum_{i=1}^{1000} i = 500500.$$

Dette svaret kunne vi også ha funnet forholdsvis enkelt ved regne ut gjennomsnittet av alle tallene og gange med antall tall:

$$S = \frac{1 + 1000}{2} \cdot 1000 = 500500.$$

Flott! Svarene våres er enige. Som tyder på at koden vår gjorde akkurat det vi ville at den skulle gjøre. Nå kan vi regne ut et par summer ved hjelp av datamaskin som er langt vanskligere å regne ut for hånd.

La oss først se på den samme summen, men regne ut kvadratet av tallene, altså

$$S = \sum_{i=1}^{1000} i^2 = 1 + 4 + 9 + 16 + \ldots + 1000^2.$$

For å finne denne summen kan vi bruke nesten identisk kode som tidligere, vi må bare endre uttrykket inne i løkka

```
S = 0
for i in range(1,1001):
    S += i**2

print S
```

som gir svaret

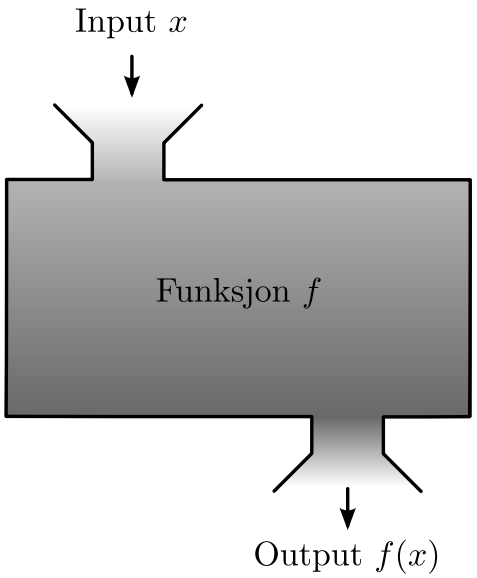
$$S = \sum_{i=1}^{1000} i^2 = 333833500.$$

Denne summen var det altså like lett å finne numerisk, men for hånd er det langt vanskligere en den enklere summen.

## 2.2 Funksjoner

Du er kanskje vandt med navnet funksjonfra matematikk. Vi skal nå vise hvordan vi kan definere funksjoner i Python. Funksjoner i programmeringssammenheng er noe bredere enn matematiske funksjoner, men vi kommer fort til å se at de har mye til felles.

Den enkleste måten å tenke på en funksjon, er å se på det som en maskin, som tar noe input, for eksempel et tall, og så gir noe output, bestemt av inputten.




Hvis vi for eksempel ser på den matematiske funksjonen

$$f(x) = x^2 + 3x + 1.$$

Så kan vi for hver verdi av $x$ (input) beregne en resulterende verdi av $f(x)$ (output). Med andre ord er funksjonen $f$ en slags regel, eller maskin, som behandler et tall vi gir den. Vi kan definere denne funksjonen i Python som følger

```
def f(x):
    return x**2 + 3*x + 1
```

her er `def` og `return` Python-kommandoer som vi skal forklare litt mer i detalj snart. Kort fortalt definerer vi her at det skal finnes en funksjon som heter $f$, som tar et tall $x$ inn, og gir tilbake (returnerer) tallet $f(x)$. Vi kan bruke funksjonen, vi kaller dette gjerne å kalle på funksjonen", som følger

```
print f(2)
print f(3.5)
print f(-1) + f(1)
```

som gir:

```
11
23.75
4
```

Så fort vi har definert en funksjon i python, huskes denne helt til programmet er ferdig å kjøre, og vi kan bruke den så mye vi ønsker. Funksjoner vi definerer, er egentlig bare en ny type variabel.

En funksjon i python, trenger ikke nødvendigvis å være matematisk. Vi kan for-eksempel lage en funksjon som følger

```
def greet(name):
    print "Hello " + name + "!"
```

Denne funksjonen tar et navn som input, det vil si, en tekst-streng, og skriver ut en hilsen som output. Kommandoen

```
greet("Lucy")
```

resulterer altså i

```
Hello Lucy!
```

Merk at denne funksjonen ikke brukte kodenavnet `return`, og når vi kallet på funksjonen, skrev vi ikke `print` før funksjonskallet. Dette er fordi funksjonen i seg selv printet, det var det vi hadde *definert* at den skulle gjøre. Det er kanskje litt vanskelig å skjønne denne forskjellen, så la oss se på et par eksempler til.

Vi definerer to funksjoner, $f_1$ og $f_2$. Vi vil at de begge skal ta et tall $x$ som input, og regne ut $2x$, altså det dobbelte. Forskjellen skal være at $f_1$ returnerer resultatet, mens $f_2$ printer det. Vi har altså:

```
def f1(x):
    return 2*x

def f2(x):
    print 2*x
```

La oss nå prøve å kalle på $f_1$ og $f_2$ på forskjellige måter og prøve å forstå hva som skjer. Først skriver vi:

```
f1(2)
```

Vi får ikke noen feilmelding, så det virker greit. Men vi får heller ingen utskrift, det skjer ingenting! Dette er fordi vi kaller på $f_1$ med tallet 2 som input, funksjonen regner ut at $2 * 2 = 4$, og returnerer verdien, men så gjør vi ingenting med denne verdien. Vi kunne for eksempel gjort

```
a = f1(2)
print a
```

Her lagrer vi den returnerte verdien i en variabel `a`, og så skriver vi ut `a`. Nå får vi resultatet til skjerm, som er 4, flott!

La oss nå prøve

```
f2(3)
```

Dette fungerer veldig fint, vi får resultatet 6, rett til skjerm, flotte saker. Dette er fordi vi kaller på funksjonen $f_2$, som skriver tallet rett til skjermen. Om vi nå derimot prøver å lagre resultatet i en variabel

```
a = f2(3)
print a
```

får vi et litt mystisk resultat:

```
6
None
```

For å skjønne hva som skjer her, så må vi først tolke kodelinjen `a = f2(3)`, som vi lærte forrige uke, så betyr en slik linje at vi skal regne ut det som er på høyre-siden, og lagre det i variabelen `a`. Vel, på høyre side kaller vi på $f_2$ med tallet $x = 3$, $f_2$ gjør som vi har definert å skriver ut resultatet $2 * x = 6$ rett til skjerm. Etter det er $f_2$ ferdig, men den har ikke *returnert* noen verdi, så når `a` settes lik resultatet på høyre-siden, så blir den ingenting, eller `None` som det heter i Python.

Du har forhåpentligvis fått en viss idé om hva det nå betyr at en funksjon returnerer en verdi ved hjelp av `return`-kommandoen. Ikke få panikk om du synes dette er ganske forvirrende, forståelse kommer med tid i programmering, så du skjønner det nok bedre etter du har fått prøvd deg litt frem!

### 2.2.1 Funksjoner av flere variabler

Når man først vet hvordan man lager funksjoner i python, er det superenkelt å lage funksjoner av flere variabler. Vi kan for eksempel lage følgende funksjon

$$f(x, y) = 2x^2 + xy + 3,$$

som følger

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

print f(3,4)
```

Tilsvarende kan vi lage funksjoner som ikke tar noen argumenter. Disse er kanskje mer nyttig i en programmeringssammenheng enn i en matematisk sammenheng. Vi kan foreksempel lage en funksjon

```
def greet():
    print "Hey there! I hope you have a great day!"
```

Merk at for å kalle på en slik funksjon, må vi fortsatt bruke parantesene, slik at et kall på `greet` skrives

```
greet()
```

En ting det er verdt å merke seg er at mange av kommandoene vi har brukt i Python hitil, er funksjoner som er definert på akkurat den måten vi har lagt frem nå. For eksempel er `range` en funksjon, som vi kaller på når vi bruker. Når vi skriver; `range(1,10,2)` så gjør vi et funksjonskall med 3 inputtall.

## 2.3 If/Else

Av og til ønsker vi at programmet vårt skal gjøre forskjellige ting basert på betingelser (*conditions* på engelsk) vi gir det.

For eksempel, hva om vi ønsker å skrive ut en spesiell beskjed dersom et tall er lik 0? For å gjøre dette, trenger vi det vi kaller en `if`-test. La oss se på et eksempel

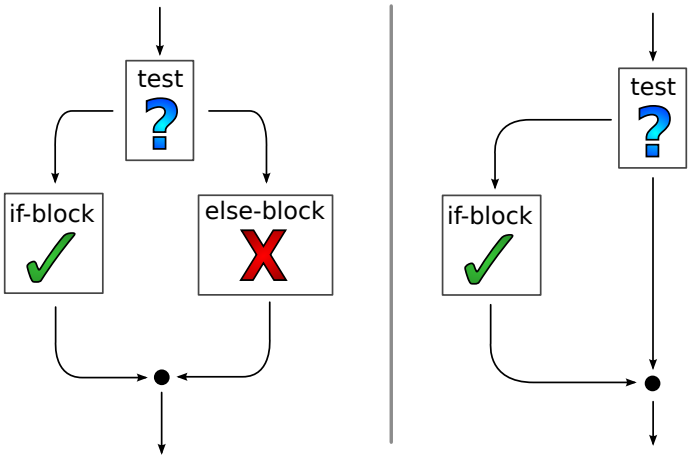
```
number = 1

if number == 0:
    print "The number equals zero!"
else:
    print "The number does not equal zero!"
```

Kjør programmet, hvilken beskjed får du? Prøv å endre `number` til 0 og se om beskjeden endrer seg.

Legg merke til at vi bruker to likhetsteng, '==’. Dette kommer av at vi allerede bruker et enkelt '=’ for å definere variabler, så `number = 0` setter variabelen `number` til å ha verdien 0.

Som du kan se ser den grunnleggende strukturen til en if/else-test slik ut:

```
if condition:
    do some things
else:
    do some other things
```




Du trenger ikke alltid en `else`-blokk. Kanskje du bare vil skrive en beskjed hvis tallet er lik 0, og ikke gjøre noe hvis ikke.

```
if number == 0:
    print "The number equals zero!"
```

```
The number does not equal zero!
```

Du kan bruke if-tester til å sjekke andre ting en likhet. Noen andre nyttige operatorer er større enn (`>`), mindre enn (`<`), større enn eller lik (`>=`), og mindre enn eller lik (`<=`). For å se om noe er `True` (sant) eller `False` (usant), kan du skrive betingelsen til skjermen:

```
print 7 > 5
print 7 >= 7
print 7 < 5
print 7 <= 5
```

Lek med litt med if/else-tester, prøv forskjellige kombinasjoner og gjett hva resultatet blir før du kjører programmet!

Du kan legge til flere betingelser med `elif`-kommandoen:

```
number = -1
if number == 0:
    print "The number equals zero!"
elif number > 0:
    print "The number is positive"
else:
    print "The number is negative"
```

## 2.4 Arrays

Vi skal snart gå inn på plotting i Python, som vil si å lage figurer. Men da bør vi først nevne `arrays`. Arrays er en spesiell type liste ment for matematikk. I motsetning til lister, som kan inneholde forskjellig type innhold, så kan arrays bare inneholde tall. Lister kan også gjøres større og mindre ved å legge til eller slette elementer, mens arrays *alltid* har samme antall elementer. Om vi lager en array med tusen tall, så vil den arrayen alltid ha tusen tall, vi kan derimot endre hvilke tall den inneholder.

Dere skal nå få se de to vanligste måtene å lage arrays på. Først, et tomt array. Ettersom at et array alltid har likt antall plasser, må vi spesifisere størrelsen på arrayet. Vi bruker kommandoen `zeros`:

```
x = zeros(3)
```

Variabelen `x` er nå et array, med tre elementer. Der alle er satt til tallet 0. Dette virker kanskje litt rart å gjøre, men vi kan nå endre spesifikke elementer ved å indeksere på følgende måte

```
x[0] = 10
x[1] = 4
x[2] = 3
```

Firkantparantesene kalles "indeksering", og brukes for å få tilgang til enkelt elementer av et array eller en liste. Python starter å telle på 0, så `x[0]` er det første elementet, og `x[1]` det andre, osv. Om vi nå skriver `print x` får vi nå resultatet

```
[ 10.   4.   3.]
```

Neste måte lage et array på, er med funksjonen `linspace`, som står for *linear spacing*. Den tar tre inputparamtere: start, stop, antall.Om vi for eksempel skriver

```
x = linspace(0,10,11)
print x
```

får vi

```
[ 0.   0.2  0.4  0.6  0.8  1. ]
```

Altså er `x` et array med 6 elementer, hvor det første elementet er 0, det siste 1, og de resterende er jevnt fordelt. Vi kommer til å se at linspace er veldig praktisk når vi skal plotte.

### 2.4.1 Vektoriserte funksjoner

En stor fordel med arrays, er at de er laget for å drive med matematikk. Arrays oppfører seg foreksempel akkurat som vektorer. Det betyr at vi kan for eksempel bruke arrays til å regne prikkprodukt og kryssprodukt

```
u = array([1,-4,3])
v = array([3,2,-1])
print dot(u,v)
print cross(u,v)
```

```
-8
[-2 10 14]
```

En annen veldig nyttig funksjonalitet, er at vi kan kalle på funksjoner med arrays som input, om vi for eksempel har laget funksjonen som vi så på tidligere

```
def f(x):
    return x**2 + 3*x + 1
```

så kan vi kalle på denne med et array som følger

```
a = array([0,1,2,3,4,5])
print f(a)
```

```
[ 1  5 11 19 29 41]
```

Det som skjer når vi kaller på funkjsonen med et array, er at Python regner ut resultatet element for element og returnerer et array med resultatene tilbake.

## 2.5 Plotting

Vi skal nå se på plotting i Python, som vil si å lage enkle figurer og grafer. Vi kommer til å tegne inn grafen vår i et koordinatsystem som vi er vant med fra matematikk. For å plotte bruker vi funksjonen `plot` fra pakken Pylab, som tar som input to lister, eller arrays, av tall. Vi kan altså for eksempel skrive

```
plot([0,0.5,1], [2,4,6], 'x')
show()
```

Her tegnes altså punktene (0,2), (0.5,4) og (1,6) inn i koordinatsystemet. Vi må bruke kommandoen `show()` for å vise figurer vi har laget. Vi har også lagt inn tekstrengen `'x'` i plotte-kommandoen, det er for at den skal tenge punktene vi gir som kryss. Hvis vi ikke ber den tegne kryss, tegner den rett og slett rette streker mellom punktene.

Om vi har definert en funksjon, for eksempel:

$$f(x) = x^2 + 3x + 1,$$

som vi har sett på tidligere. Kan vi nå gjøre som følger

```
def f(x):
    return x**2 + 3*x + 1

x = linspace(-6,6,1000)
y = f(x)

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

Her lager vi altså et sett med tusen punkter, som vi så plotter. Vi får dermed en fin figur av funksjonen $f(x)$.

Tilsvarende kan vi lage plot av velkjente matematiske funksjoner, for eksempel sinus og cosinus.

```
x = linspace(0,2*pi,1000)
plot(x,sin(x))
plot(x,cos(x))
show()
```

Merk at siden vi ga to plotte kommandoer før vi brukte show, så får vi to kurver i samme figur.

Etter vi har lagd kurven ved å bruke `plot`-kommandoen, og før vi bruker `show`, så kan vi pynte på figuren vår. Vi kan for eksempel legge til navn på akser med

```
xlabel('x')
ylabel('y')
```

Vi kan lage tittel på figuren med `title`-funksjonen på samme måte. Vi kan definere hvilke deler av figuren vi skal vise med `axis`, for eksmepel

```
axis([0,2*pi,-1,1])
```

vi gir altså en liste med `[xstart, xstop, ystart, ystop]`.

Vi kan også lagre figuren vår med

```
savefig('figure1.png')
savefig('figure1.pdf')
```

som lagrer bilde som filene 'figure1.png' og 'figure2.pdf' henholdsvis.

Det er mange andre muligheter for å pynte på plot og få dem til å se kule ut, men la oss ikke dykke for dypt inn i det akkurat nå. Vi kommer til å se mer på plotte-muligheter iløpet av ukene som kommer, men om du er utålmodig kan du se på `matplotlib.org` som er nettsiden til plottepakken som pylab bruker, der finnes det mange eksempler på plots man kan lage.

## 2.6 Oppgaver

### 2.6.1 Oppgave 2.1

(a) Definer en funksjon `celsius_to_fahrenheit` som tar grader $C$ som input, og returnerer grader Fahrenheit. Vi minner om at formelen for omregningen er

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

(b) Bruk funksjonen til å finne ut hvor mange grader Fahrenheit disse tempraturene er: $20°$, $0°$, $-40°$.

(c) Skriv en løkke der du lar gradier i Celsius gå fra -100 til 100 og skriv ut de tilhørende gradene i Fahrenheit ved å bruke funksjonen din.
**Hint 1:** Bruk `range`-funksjonen
**Hint 2:** Du må kalle på funksjonen din inne i løkka.

### 2.6.2 Oppgave 2.2

Ta en titt på følgende program

```
def sumopptil(stop):
    s = 0
    for i in range(1,stop+1):
        s += i
        return s

print sumopptil(10)
```

(a) Programmet skal regne ut summen av alle tall fra 1 opp til 10. Regn ut det korrekte svaret for hånd eller med kalkulator. Kjør koden og sammenlign svarene.

(b) Finn feilen i programmet og fiks det slik at det gir rett svar
**Hint 1:** Det kan være lurt å gå igjennom programmet trinn for trinn som en datamaskin og skrive ned hva `s` er for hvert steg i programmet.

### 2.6.3 Oppgave 2.3

Vi skal nå bruke løkker og `range`-funksjonen til å regne ut noen matematiske summer

(a) Regn ut summen av alle oddetall under 1000.

(b) Regn ut summen av alle kvadrattall til og med 10000.
**Hint:** Vi er altså ute etter summen

$$1 + 4 + 9 + 16 + \ldots + 10000 = \sum_{i=1}^{100} i^2.$$

(c) Regn ut summen av alle kubetall fra 8 til og med 1000.

(d) Regn ut summen av den uendelige rekka

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots$$

**Hint:** Hvert ledd i summen blir mindre og mindre, det holder altså å ta med for eksempel de 1000 første leddene. Da alle ledd etter dette vil bidra ekstremt lite til det endelige svaret

### 2.6.4 Oppgave 2.4

(a) Plot $e^x$ for $x \in [0, 2]$.

(b) Lag et plot, der du viser de tre funksjonene:

$$e^x, \qquad e^{-x}, \qquad 1/e^x.$$

for $x \in [0, 2]$. Her må du bruke `axis`-kommandoen for å velge rimelige akser på figuren din!

(c) Pynt på plottet du nettopp lagde ved hjelp av funksjonene `axis`, `xlabel`, `legend`, `title` og `grid`.

(d) Lagre plottet ditt som en .pdf-fil, og som en .png-fil. Sjekk at filene ble lagret riktig og at de ser ut som forventet.

(e) Definer en funksjon for den matematiske funksjonen

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

Plot $f(x)$ for x $x \in [0, 5]$. Gjør plottet pent og lagre det som en .png-fil

## 2.7 Utfordringer!

### Primtall

Greier du å skrive en funksjon som tar et heltall som input, og finner ut om det er et primtall eller ikke?

### Fibonacci

Greier du å skrive et program som regner ut og skriver ut Fibonacci-tallene?

### Project Euler

På nettsiden `www.projecteuler.net` er det mange morsomme matematiske nøtter som er ment å løses ved programmering. Se om du greier å få til et par av de første oppgavene. Spør gjerne om hjelp.

# 3 Uke 3

Denne uka skal vi bruke det vi lærte i uke 1 og 2 til å se på populasjonsutvikling. Før vi kan begynne med programmeringa må vi regne litt og komme frem til noen m*matematiske modeller*.

## 3.1 Tallfølger

Tall som kommer etter hverandre i en bestemt rekkefølge, kaller vi en tallfølge. La oss se på tallfølgen

$$1, 3, 5, 7, 9$$

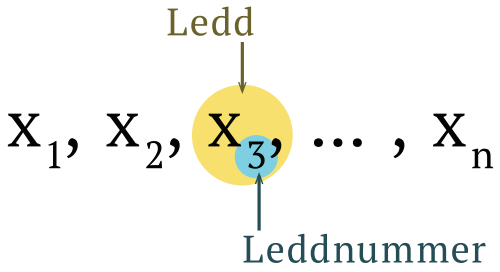
Denne tallfølgen består av de fem første oddetallene i stigende rekkefølge. Vi kaller det for en *endelig* tallfølge med fem *ledd*. Dersom vi har en lengre tallfølge bruker vi gjerne "..." for å slippe å skrive alle leddene i rekka. For eksempel kan vi skrive følgen av alle oddetall under 100 slik:

$$1, 3, 5, 7, 9, \ldots, 99$$

"..." signaliserer at resten av leddene skal følge samme mønster. Her får vi f.eks. neste tall ved å legge til 2. Hvis vi vil ha en *uendelig* tallfølge, for eksempel alle oddetall, kan vi skrive det slik:

$$1, 3, 5, 7, 9, \ldots$$

Det er vanlig å kalle første ledd for $x_1$, andre ledd for $x_2$, og så videre. Indeksene $1, 2, 3, \ldots$ forteller hvilket nummer i tallfølgen leddet har. $x_n$ er med andre ord ledd nr. $n$ tallfølgen. I følgen over har vi da at $x_1 = 1$, $x_2 = 3$, $x_3 = 5$, $x_4 = 7$ osv.




**Figur 1:** Leddnummer

### 3.1.1 Rekursive formler

Rekursive formler beskriver hvordan det neste leddet i tallfølgen avhenger av leddet foran. En slik formel lar oss altså regne ut $x_n$ når $x_{n-1}$ er kjent.

**Eksempel**

Vi har formelen $x_n = x_{n-1} + 2$ Hvilken følge er dette når det første leddet er 1? Hva er de fire første leddene?

Formelen sier at vi kommer fra et ledd til neste ved å legge til 2

$$\begin{aligned} x_1 &= 1 \\ x_2 &= x_1 + 2 = 1 + 2 = 3 \\ x_3 &= x_2 + 2 = 3 + 2 = 5 \\ x_4 &= x_3 + 2 = 5 + 2 = 7 \\ &\vdots \end{aligned}$$

Når det første leddet, $x_1$, er 1 har vi altså at den rekursive formelen $x_n = x_{x-1}+2$ beskriver oddetallene som vi så på tidligere.

## 3.2 Oppgaver:

### 3.2.1 Oppgave 3.1

a) Hvilken følge får vi fra $x_n = x_{n-1} + 2$ når $x_1 = 2$?

b) Gitt den rekursive formelen $x_n = 2x_{n-1}$ for $x_1 = 1$, finn de fire første leddene.

## 3.3 Modellering med rekursive formler

La oss se på et veldig enkelt eksempel. Gitt at du har en avtale om at du får en krone i lommepenger hver dag. Dette kan beskrives med følgende rekursive formel:

$$x_n = x_{n-1} + 1$$

Hvis du vet hvor mange penger du har i dag, $x_{n-1}$, kan du altså forutse hvor mange penger du kommer til å ha i morgen. Det er bare å legge til en krone. For å finne ut hvor mange penger du kommer til å ha i overmorgen gjentar du bare det samme en gang til. Du kan altså simulere hvor mange penger du kommer til å ha langt inn i fremtiden. Så lenge pengeveksten din fortsetter å følge denne modellen, det vil si at du hverken bruker penger eller mottar penger på noen annen måte.

**Eksempel**

Sett at vi har 100 kroner som vi setter inn i en bank med 10% årlig rente, dvs. at vi på slutten av året får 10% av vi pengene vi for øyeblikket har liggende i

banken. Anta at vi aldri legger inn eller tar ut noen penger fra banken, hvordan kan vi modellere hvor mange penger du kommer til å ha fremover?

Anta at vi vet hvor mange penger du har ved år $n-1$, dvs. at vi kjenner $x_{n-1}$. Kan vi finne et utrykk for hvor mange penger vi kommer til å ha ved år $n$? Vi vet at vi kommer til å beholde de $x_{n-1}$ pengene vi hadde fra før. I tillegg får vi 10% av de pengene i renter. Altså får vi $0.1x_{n-1}$ nye penger. Dette gir oss formelen

$$x_n = x_{n-1} + 0.1x_{n-1}$$

Hvis vi nå for eksempel vil vite hvor mange penger vi kommer til å ha om 4 år er dette gitt ved:

$$\begin{aligned}
x_1 &= 100 \\
x_2 &= x_1 + 0.1 \cdot x_1 = 100.0 + 0.1 \cdot 100.0 = 110.0 \\
x_3 &= x_2 + 0.1 \cdot x_2 = 110.0 + 0.1 \cdot 110.0 = 121.0 \\
x_4 &= x_3 + 0.1 \cdot x_3 = 121.0 + 0.1 \cdot 121.0 = 133.1 \\
x_5 &= x_4 + 0.1 \cdot x_4 = 133.1 + 0.1 \cdot 133.1 = 146.41
\end{aligned}$$

Som du ser, kan det fort bli ganske slitsomt å skrive ut alt dersom vi vil regne ut mange ledd. Hvis vi for eksempel vil vite hvor mange penger vi har om 50 år, vil dette ta fryktelig lang tid å regne ut for hånd.

## 3.4 Oppgaver

### 3.4.1 Oppgave 3.2

a) Hvor mange penger har du i banken etter 5 år? Hva med etter 6 år?

## 3.5 Populasjonsvekst

Vi kan bruke rekursive formler til å modellere veksten i en populasjon av organismer. La oss prøve å diskutere oss fram til en brukbar modell for å beskrive veksten av en harebestand. Først ser vi på modellen vi brukte i eksemplet med lommepengene.

$$x_n = x_{n-1} + 1$$

Hvis vi bruker denne modellen til å beskrive harebestanden tilsvarer det at vi får en ny hare hver dag for alltid. Det høres ikke veldig realistisk ut. Det er naturlig å tenke seg at antall nye harer som fødes er avhengig av hvor mange harer som eksisterer. Vi kan dermed tenke oss at antall nye harer som fødes er gitt ved $by_{n-1}$, hvor $b$ er fødselsraten til harene. I tillegg lever ikke harene evig, så det vil hele tiden være noen av de eksisterende harene som dør. Antall døde

harer er også avhengig av antall harer, og er gitt ved $dx_{n-1}$ hvor $d$ er dødsraten. Formelen blir da:

$$y_n = y_{n-1} + b \cdot y_{n-1} - d \cdot y_{n-1}$$
$$y_n = y_{n-1} + (b - d) \cdot y_{n-1}$$

Vi ser at netto vekst av populasjonen er $(b - d)y_{n-1}$.Vekstraten kan uttrykkes som $r = b - d$ slik at vi får

$$y_n = y_{n-1} + r \cdot y_{n-1}$$

Da har vi funnet en modell som vi kan bruke til å modellere populasjonsveksten for så mange ledd vi trenger. Denne modellen kan vi også bruke til å beskrive mennesker, andre dyr, celler eller hvilke som helst objekter hvor antall fødsler og dødsfall er proporsjonale med antallet individer.

$n$ trenger ikke å måles i dager, men $b$ og $d$ er avhengig av hvor lang tid det går mellom hvert ledd. Det vil si at for tilsvarende populasjon vil verdiene av $b$ og $d$ være større hvis $n$ måles i år enn hvis $n$ er dager.

**Eksempel**

La oss gjøre dette litt mer konkret med et eksempel. $y$ representerer nå antall harer som bor på en øy i en innsjø. Fødselsraten er 7% per måned, dødsraten er 2% og vi starter med en bestand på 100 harer. Hvor mange harer bor det på øya etter 2 år?

Nå må vi sette dette inn i formelen vår. Da må vi først huske at

$$7\% = \frac{7}{100} \quad \text{og} \quad 2\% = \frac{2}{100}.$$

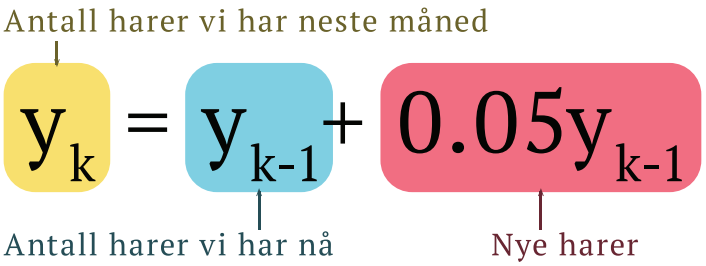
Vi har altså $b = \frac{7}{100} = 0.07$ og $d = \frac{2}{100} = 0.02$ som gir vekstraten

$$r = b - d = 0.07 - 0.02 = 0.05.$$

Videre vet vi at vi starter med $y_0 = 100$ og vi kan regne ut

$$y_1 = y_0 + r \cdot y_0 = 100 + 0.05 \cdot 100 = 105$$
$$y_2 = y_1 + r \cdot y_1 = 105 + 0.05 \cdot 105 = 110.25$$
$$\vdots \qquad \vdots$$

Slik kan vi fortsette å regne ut utviklingen av harebestanden.




**Figur 2:** Populajsonsvekstformel

Her er tidssteget mellom $y_0$ og $y_1$ én måned så vi må regne ut 24 tidssteg for å finne ut hvor mange harer som bor på øya etter to år. I stedet for å gjøre dette for hånd kan vi skrive et lite program som gjør dette for oss!

```
from numpy import *
n  = 12*2     #antall tidsintervaller
y0 = 100      #antall harer naar vi starter
r  = 0.05     #vekstrate (dvs 2% per mnd)
index_set = range(n+1)
y = zeros(len(index_set))
y[0] = y0
for k in index_set[:-1]:
    y[k+1] = y[k] + r*y[k]
```

Vi kan nå skrive ut resultatet ved å legge til

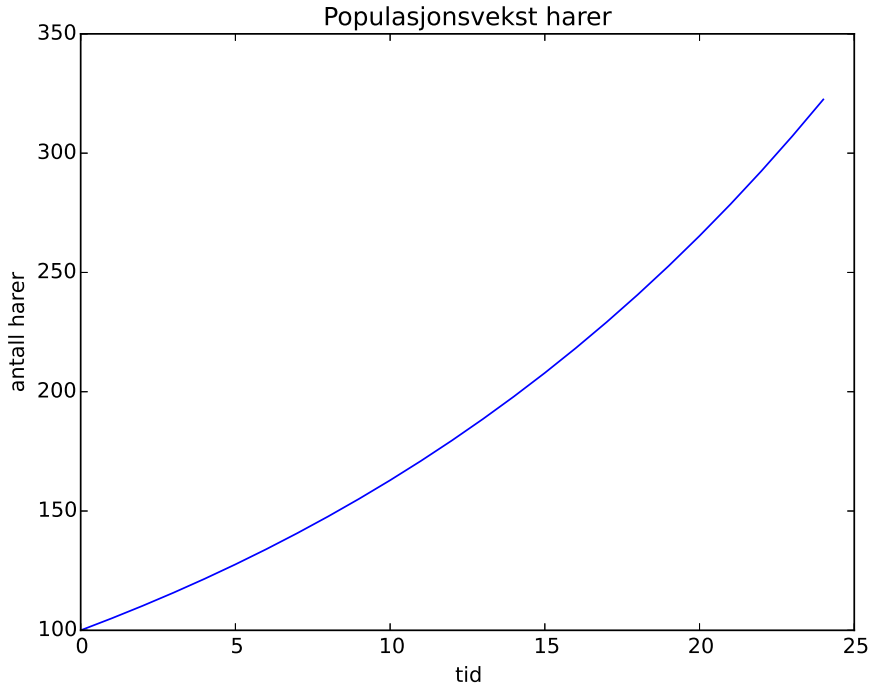
```
print y
```

nederst i programmet. Vi får da:

```
[ 100.          105.          110.25        115.7625      121.550625
  127.62815625  134.00956406  140.71004227  147.74554438  155.1328216
  162.88946268  171.03393581  179.5856326   188.56491423  197.99315994
  207.89281794  218.28745884  229.20183178  240.66192337  252.69501954
  265.32977051  278.59625904  292.52607199  307.15237559  322.50999437]
```

For å få en bedre idé om hva dette betyr kan vi plotte utviklingen.

```
from matplotlib.pyplot import *
plot(index_set, y)
xlabel('tid')
ylabel('antall harer')
title('Populasjonsvekst harer')
show()
```




**Figur 3:** Output plot

## 3.6 Oppgaver

### 3.6.1 Oppgave 3.3

a) Hva skjer med harebestanden etter 5, 10 og 20 år?

b) La oss si at det dukker opp en ørn i område og dødsraten øker til 20% per mnd. Hva skjer med harebestanden etter 2 år?

c) Diskuter med sidemannen. Hva syns dere om denne modellen? Når kan den være nyttig? Når er den urealistisk?

### 3.6.2 Oppgave 3.4

a) Lag et program som simulerer vekst av bakterier. Vi starter med en bakterie. Antall bakterier dobles hver time. Hvor mange bakterier har vi etter ett døgn?

b) Bakteriene fra a) dyrkes i en bakteriekultur. Etter 15 timer begynner det å bli lite plass og mat igjen og det dør like mange bakterier som det skapes ved celledeling. Etter 18 timer begynner antall celler å synke og $r = -0.3$. Tilpass programmet du laget i a) og finn ut hvor lang tid det tar før antall bakterier er mindre enn 1000. **Hint:** Bruk if/else-test.

Du kan lese mer her: http://www.textbookofbacteriology.net/growth_3.html

# 4 Uke 4

Forrige uke så vi på noen modeller for populasjonsutvikling som ikke var realistiske for lange tidsperioder. Denne uka skal vi derfor se på modeller som tar for seg bærekraftig vekst og hvordan populasjonsutviklingen er i et økosystem med et rovdyr og et byttedyr.

## 4.1 Bærekraftig vekst

Den forrige modellen førte til at bestanden vokste raskere og raskere ettersom tiden gikk. Denne typen vekst kaller vi *eksponensiell vekst*, og en slik type vekst kan ikke fortsette for alltid. I naturen er det som regel en øvre grense ($M$) for hvor mange individer som kan leve i et område samtidig. Mat- og plassmangel, konkurranse mellom individer, rovdyr og smittsomme sykdommer er alle faktorer som begrenser veksten av en bestand. Tallet $M$ kalles systemets *bæreevne* og representerer den maksimale størrelsen på en bestand som er bærekraftig over tid. Når veksten er begrenset betyr det at vekstraten $r$ må være en funksjon avhengig av tiden:

$$y_n = y_{n-1} + r(n-1) \cdot y_{n-1}$$

I begynnelsen, når det er nok ressurser, så vokser bestanden eksponensielt. Men med tiden, når $y$ går mot $M$, stopper veksten opp og $r$ går mot null. Her er en funksjon med disse egenskapene

$$r(n) = R \cdot \left(1 - \frac{y_n}{M}\right)$$

Observer at når $n$ er liten så er $y_n$ veldig mye mindre enn $M$ (befolkningen har så vidt begynt å vokse). Da blir $\frac{y_n}{M}$ fryktelig liten, nesten 0. Dette betyr at $1 - \frac{y_n}{M}$ blir veldig nærme 1. For **liten n** har vi altså at

$$r(n) \approx R \cdot \ (1 - 0) = R.$$

Altså er vekstraten **R** når det er **ubegrenset tilgang til resurser**. Men når $y_n$ (antall individer) nærmer seg $M$ så blir $\frac{y_n}{M} \approx 1$. Da har vi at

$$r(n) \approx R \cdot (1 - 1) = 0,$$

så når $y_n$ nærmer seg $M$, går vekstranten $r(n)$ mot null akkurat slik som vi ønsket. Dette kalles *logistisk vekst* og kan skrives som en likning på formen

$$y_n = y_{n-1} + Ry_{n-1}\left(1 - \frac{y_{n-1}}{M}\right)$$

### Eksempel

La oss gå tilbake til harene fra forrige eksempel. Fortsatt representerer $y$ antall harer som bor på en øy. Til å begynne med er fødselsraten 7% per måned, dødsraten 2% og vi starter med en bestand på 100 harer. Vi antar at for at bestanden skal være bærekraftig kan det maksimalt være 500 harer på øya. Hvor mange harer bor det på øya etter 5 år?

Vi har altså $b = 0.07$ og $d = 0.02$ som gir initiell vekstrate $R = b - d = 0.05$. Videre vet vi at vi starter med $y_0 = 100$ og at bæreevnen $M = 500$. Nå kan vi sette dette inn i den nye formelen vår og regne ut

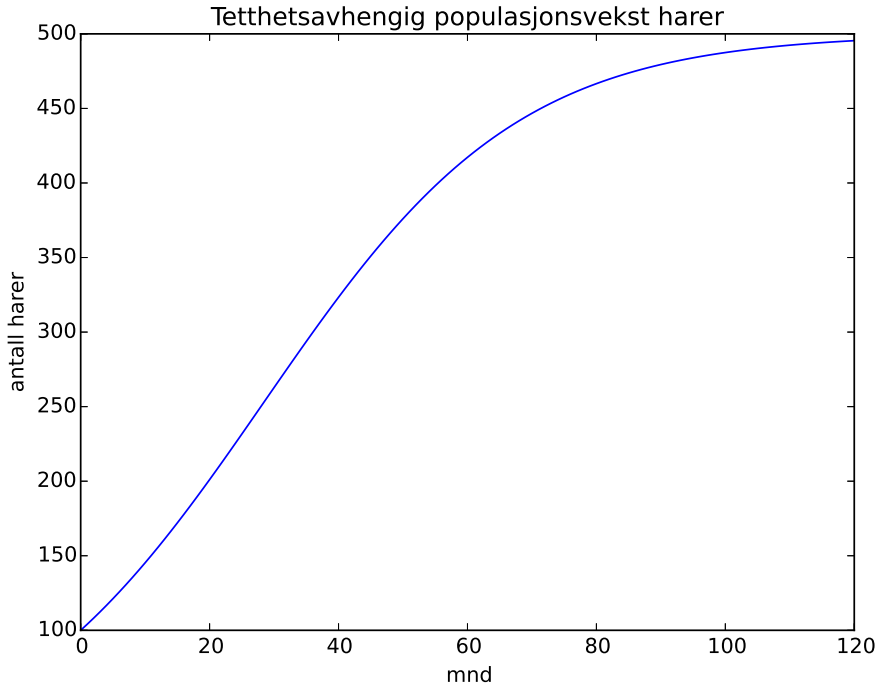
$$y_1 = y_0 + Ry_0\left(1 - \frac{y_0}{M}\right) = 100 + 0.05 \cdot 100\left(1 - \frac{100}{500}\right) = 104$$

$$\vdots \qquad \vdots$$

Slik kan vi fortsette å regne ut utviklingen av harebestanden. Her er tidssteget mellom $y_0$ og $y_1$ en måned så vi må regne ut 60 tidssteg for å finne ut hvor mange harer som bor på øya etter fem år. I stedet for å gjøre dette for hånd kan vi skrive et tilsvarende program som i forrige ukes eksempel:

```
from numpy import *
from matplotlib.pyplot import *

y0 = 100        # antall individer naar vi starter
M = 500         # baereevne
R = 0.05        # vekstrate naar vi starter
N = 12*10       # antall tidsintervaller
index_set = range(N+1)
y = zeros(len(index_set))

#Regn ut loesning for hvert intervall
y[0] = y0
for n in index_set[1:]:
    y[n] = y[n-1] + R*y[n-1]*(1 - y[n-1]/float(M))
print y

#plot resultat
plot(index_set, y)
xlabel('mnd')
ylabel('antall harer')
title('Tetthetsavhengig populasjonsvekst harer')
show()
```

Resultatet er plottet i fig 4.




**Figur 4:** Utvikling av harebestand over 10 år.

## 4.2 Oppgaver

### 4.2.1 Oppgave 4.1

a) Hva skjer med bestanden etter 100 år? Hva med etter 200 år?

b) Hva skjer hvis vi endrer vekstraten til $r = 1.5$. Hva betyr egentlig dette?

c) Diskuter hvor realistisk denne modellen er.

d) Hva tror du skjer dersom du starter med flere individer enn bæreevnen tillater (f.eks. $y_0 = 600$)? Skriv et program for å se om du hadde rett.

### Oppgave 4.2

La $x$ representere antall reinsdyr som bor på en øy. Vi starter med 50 reinsdyr, fødselsraten er 5% per måned og dødsraten er 2%. Vi antar at for at bestanden skal være bærekraftig kan det maksimalt være 600 reinsdyr på øya.

La oss i tillegg si at etter 15 år minsker ressursene på øya dramatisk og det er nå bare plass til 200 reinsdyr for at bestanden skal fortsette å være bærekraftig. Lag et plot som viser utviklingen i en slik situasjon over 30 år. **Hint:** Dette kan løses enten med if/else eller med to for-løkker

## 4.3 Rovdyr og byttedyr

La oss se på en tredje populasjonmodell: *Byttedyr-rovdyr modellen*. Denne modellen brukes til å beskrive samspillet mellom en rovdyrbestand og en byttedyrbestand. Det finnes flere slike par i naturen, for eksempel løver og gaseller eller gauper og harer.

La bruke gauper og harer som et eksempel. La mengden gauper etter $n$ måneder være gitt ved $x_n$, og mengden harer etter $n$ måneder være gitt ved $y_n$. Antall gauper er avhengig av hvor mange harer de kan finne og spise, mens antall harer er avhengig av hvor mange gauper som kan spise dem.

Vi kan beskrive denne sammenkoblede befolkningsveksten med følgende likninger:

$$y_n = y_{n-1} + cy_{n-1} - dx_{n-1}y_{n-1}$$
$$x_n = x_{n-1} - ax_{n-1} + bx_{n-1}y_{n-1}$$

hvor $a$, $b$, $c$, og $d$ er positive konstanter.

**Den første likningen beskriver utviklingen av harebestanden**

- '$y_{n-1} + cy_{n-1}$'-leddet tilsvarer veksten av harer når det ikke er noen gauper til å spise dem. Da vokser harebestanden likt som den eksponentielle modellen vi så på tidligere med vekstrate $c$.
- '$-dx_{n-1}y_{n-1}$'-leddet representerer harene som blir spist av gauper og dermed forsvinner fra bestanden med rate $d$. Det inneholder produktet av $x_n$ og $y_n$ fordi denne prosessen involverer både harer og gauper

**Den andre likningen beskriver utviklingen av gaupebestanden**

- '$x_{n-1} - ax_{n-1}$'-leddet representerer hva som skjer med gaupene dersom det ikke finnes noen harer. Da dør gaupene av sult med en rate $a$.
- ' $bx_{n-1}y_{n-1}$'-leddet beskriver hvordan det går med gaupene når de får mat og kan reprodusere med en rate $b$.

Denne modellen kan beskrive flere utfall for byttedyrene og rovdyrene. Utryddelse, likevekt og rovdyr-byttedyr-syklus.

### Eksempel

La oss se på et konkret eksempel. Antar at vi har en øy som er bebodd av både harer og gauper. Antall gauper etter $n$ måneder er gitt ved $x_n$, og antall harer etter $n$ måneder er gitt ved $y_n$. Harene har en vekstrate på 5% og de blir spist av gauper med en rate på 0.03%. Gaupene dør av sult med en rate på 2% og reproduserer med en rate på 0.01%. Lag et plot som viser befolkningsveksten til de to artene når $x_0 = 50$ og $y_0 = 100$.

$$y_n = y_{n-1} + 0.05y_{n-1} - 0.0003x_{n-1}y_{n-1}$$
$$x_n = x_{n-1} - 0.02x_{n-1} + 0.0001x_{n-1}y_{n-1}$$

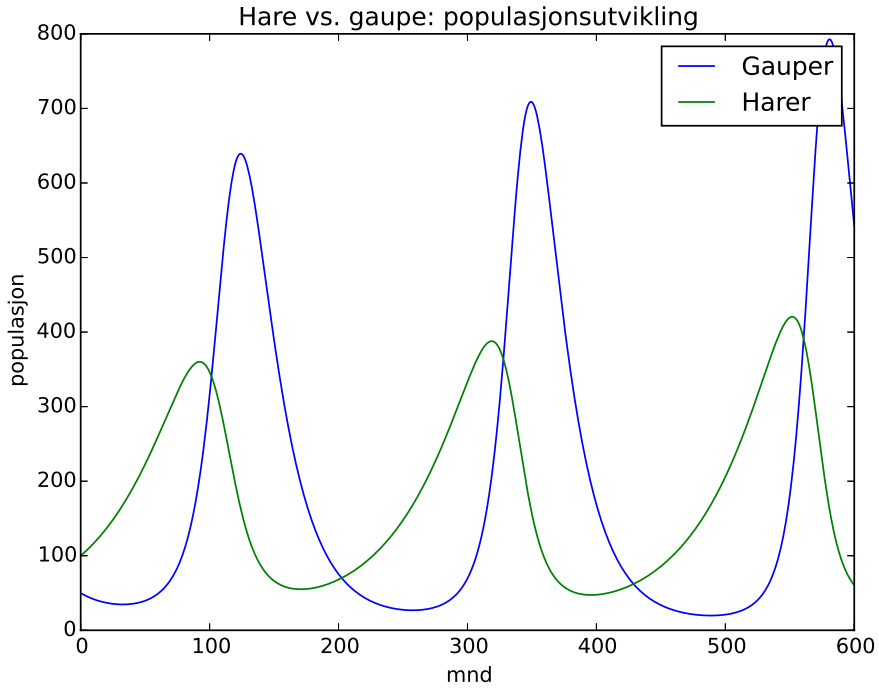
$$y_1 = y_0 + 0.05y_0 - 0.0003x_0y_0 = 100 + 0.05 \cdot 100 - 0.0003 \cdot 50 \cdot 100 = 103.5$$
$$x_1 = x_0 - 0.02x_0 + 0.0001x_0y_0 = 50 - 0.02 \cdot 50 + 0.0001 \cdot 50 \cdot 100 = 49.5$$

$$\vdots \quad \vdots$$

Vi lar datamaskinen gjøre resten av utregningene for oss i dette programmet:

```
from numpy import *
from matplotlib.pyplot import *
n = 12*50     #antall tidsintervaller
y0 = 100      #antall byttedyr naar vi starter
x0 = 50     #antall rovdyr naar vi starter
index_set = range(n+1)

x = zeros(len(index_set))
y = zeros(len(index_set))


a = 0.05   # sultrate gauper
b = 0.0003 # reproduksjonsrate gauper

c = 0.02   # vekstrare harer
d = 0.0001 # spistrate harer


y[0] = y0
x[0] = x0
for k in index_set[:-1]:
    #print y[k]
    y[k+1] = y[k] + c*y[k] - d*y[k]*x[k]
    x[k+1] = x[k] - a*x[k] + b*x[k]*y[k]

plot(index_set, x)
plot(index_set, y)
legend(["Gauper","Harer"])
title('Hare vs. gaupe: populasjonsutvikling')
xlabel('mnd')
ylabel('populasjon')
show()
```

Resultatet er vist i fig 5.




**Figur 5:** Utvikling av hare- og gaupebestand over 50 år.

## 4.4 Oppgaver

### 4.4.1 Oppgave 4.3

a) Hvor mange harer lever etter 10 år? hvor mange gauper?

b) Hva skjer hvis vi endrer vekstraten til harer til $c = 0.005$. Hva betyr egentlig dette?

c) Diskuter hvor realistisk denne modellen er.

### 4.4.2 Oppgave 4.4

a) La $y_n$ være antall lemen som bor på en øy sammen med rever gitt med $x_n$. Vekstraten til lemenene er 6% og de blir spist av gauper med en rate på 0.03%. Revene dør av sult med en rate på 2% og reproduserer med en rate på 0.01%. Lag et plot som viser befolkningsveksten til de to artene når $x_0 = 100$ og $y_0 = 50$

b) Prøv forskjellige startverdier for antall lemen og rev.

c) Sett $c = 0$. Hva skjer? Diskuter med sidemanen hva du tror dette betyr.