Introduksjon til programmering
Bjørnegård Skole

Jonas van den Brink

j.v.d.brink@fys.uio.no

June 19, 2015

> *"Everybody in this country should learn to program a computer... because it teaches you how to think."* –Steve Jobs

Dette skrivet hører til et introduksjonskurs ved Bjørnegård skole. Over 12 timer skal jeg prøve å gi dere en introduksjon til programmering. På så kort tid kan jeg selvfølgelig ikke lære dere alt, men jeg skal forhåpentligvis få vist dere en del av hovedpoengene i programmering. Idéen er det skal bli lettere for dere å lære mer i fremtiden, og det er et av målene mine med dette kurset.

Det er ingen tvil at programmering blir mer og mer viktig. Internet er såvidt 20 år gammelt, og datamaskiner litt over 60 år gammelt. Idag er vi omringet av datamaskiner, smarttelefoner og tablets—og de blir kraftigere og kraftigere i et enormt tempo. Datamaskiner brukes mer og mer, og fler og fler jobber kommer til å innehold hvertfall en del programmering. I fremtiden er jeg veldig sikker på at programmering kommer til å undervises i skolen på lik linje med matematikk.

Men det er ikke bare fordi programmering er viktig og nyttig jeg ønsker å lære dere det, det er også utrolig morsomt. Når man programmerer, så lager man noe. Programmering er en evne som lar deg gjøre om idéer og tanker du har til en virkelighet. Om du har en idé om en fantastisk app til smarttelefonen din, så kan du faktisk lage den appen. Det å programmere er rett og slett en utrolig kreavtiv ting å drive med, man kan nesten dra det såpass langt at man kan sammenligne det med å spille et instrument, eller å male eller tegne.

Men til slutt, en liten advarsel. Programmering er en helt ny ferdighet du skal lære, og det er utfordrende å skulle lære noe helt nytt. Det kommer nok til å være vanskelig og litt frustrende til tider, men forhåpentligvis også veldig belønnende når du greier å skrive dine første programmer. Dette prosjektet er altså en real utfordring til deg, og jeg håper du tar den på strak arm og står på. Jeg kommer til å være her hele veien for å hjelpe deg gjennom det, men til syvende og sist er det du som må gjøre arbeidet.

Hva er programmering?

Å programmere betyr rett og slett å lage dataprogrammer. Fra hverdagen er vi vandt med at dataprogrammer ofte er store og kompliserte, enten det er spill, dokumentbehandlingsverktøy som Microsfot Word, bildebehandlingsverktøy som Photoshop, eller internetbrowsere som Google Chrome eller Mozilla Firefox. Alle disse er eksempler på programmer, men disse er veldig store programmeringsprosjekter, hvor mange personer har samarbeidet for å lage et helhetlig produkt som skal kunne gjøre veldig masse forskjellig.

De fleste programmer som skrives er i motsetning til disse veldig små og spesialiserte. Grunnen til at dere kanskje ikke har hørt så mye om sånne programmer, er at de enten aldri har blitt delt. En som kan programmere skriver ofte et program som man bare bruker selv, og aldri deler med noen. Eller så kan det være at dataprogrammet bare er skjult fra dere. Tenk for eksempel på mikrobølgovnen deres hjemme, eller ovnen, eller vaskemaskinen, eller TVen deres—alle disse har blitt *programmert* til å utføre spesialiserte oppgaver. Tusenvis av dataprogrammer ligger i bakgrunnen av hverdagen deres for å gjøre ting så lette som mulig for dere, og det er jo nettopp det at dere slipper å tenke på dem, som gjør dem så geniale.

Programmeringsspråk

Å programmere handler altså om å gi datamaskinen instrukser for at den skal utføre en bestemt oppgave, eller løse et bestemt problem. Vi gir disse instruksene til datamaskinen ved hjelp av spesielle programmeringsspråk, som rett og slett er et språk datamaskinen forstår. Eksempler på programmeringsspråk som brukes mye idag er Python, Java, C++ og JavaScript. Akkurat som vanlig språk, så finnes det mange hundre forskjellige programmeringsspråk, og de har mange fellestrekk. I motsetning til vanlig språk derimot, så er det mye lettere å lære seg flere programmeringsspråk hvis man allerede kan ett. I dette kurset kommer vi til å bruke Python, som er et av de aller mest brukte programmeringsspråkene idag—hvis dere har et godt grep om Python, så kan dere ganske lett lære dere de fleste andre programmeringsspråk på kort tid.

Problemløsning

Når vi skriver instrukser i form av programmeringsspråk, så kaller vi det kode. Det handler altså om å skrive den riktige koden, sånn at datamaskinen gjør det vi vil at den skal gjøre. Det som gjør programmering vanskelig, er at datamaskiner er veldig dumme. I motsetning til oss mennesker, kan datamaskiner kun forstå veldig enkle instrukser. Før vi kan sette igang med å skrive selve koden for programmet vår, må vi altså finne ut hvordan vi kan løse oppgaven vår med de enkle instruksene datamaskinen kan forstå. Vi må *bryte ned* problemet vårt til små, håndterlige biter som datamaskinen kan få til.

Når vi bryter opp et problem på denne måten, så lager vi en slags oppskrift for hvordan et problem kan løses. Du kan tenke på det litt som et mattestykke, for å komme i mål for å løse en matteoppgave, så man første gjøre en liten ting, og så en annen liten, og så en tredje liten ting, helt til man sitter med det siste

svarer til slutt. Hvert steg i seg selv er aldri særlig vansklig, utfordringen ligger i å finne ut hvilke steg man skal ta, og i hvilken rekkefølge man skal gjøre dem. I programmering så kaller man gjerne en slik 'oppskrift' for en algoritme. Det er litt rart ord, men en algoritme er altså egentlig bare oppskriften av hvilke steg man skal ta for å komme i mål.

Datamaskiner er dumme

Siden datamaskiner er så dumme, krever det også at koden vi skriver er helt riktig. Hvis du skriver et brev til en venn, og du har et par skrivefeil i teksten din, så er ikke det så farlig, for vennen din kommer mest sannsynlig til å skjønne det du har skrevet uten problemer, kanskje vennen din ikke engang legger merke til skrivefeilene dine. Med en datamaskin derimot, så vil en enkel skrivefeil være totalt uforståelig, og mest sannsynlig vil ikke dataprogrammet ditt funke i det heletatt hvis du har en skrivefeil i koden din.

Det hørest kanskje utrolig vanskelig ut, at en skrivefeil skal ødelegge hele programmet ditt. Hvordan kan du klare å skrive kode uten å ha noen småfeil? Vel, jeg kan love deg med en gang at du ikke kommer til å få til det. Ingen får til det, ikke engang de som har skrevet kode i mange, mange år. Det er derfor en stor del av det å programmere, er å finne feilene man har gjort, å rette på dem. Småfeil i kode kalles ofte for *bugs*, og det å lete etter feil og rette dem opp kaller vi derfor for *bugfixing*. Bugfixing er en viktig del av programmering, og er mye av det man bruker tiden sin på, så det er verdt å bli flink på det. Heldigvis er datamaskinen flink til å si ifra om hva det er den ikke forstår, så bugfixing kan faktisk være ganske lett og gøy, men la oss se mer på dette når du faktisk har lært litt kode.

Lightbot

Så langt har jeg jo fortalt en del om hvordan programmering funker, men det så langt har jo alt vært veldig generelt og vagt. La oss derfor snu oss til et eksempel på programmering, et spill som heter Lightbot. Jeg syns dette spillet er en flott måte å se nærmere på prinsippene i programmering, uten at vi trenger å sette oss inn i et helt programmeringsspråk helt enda.

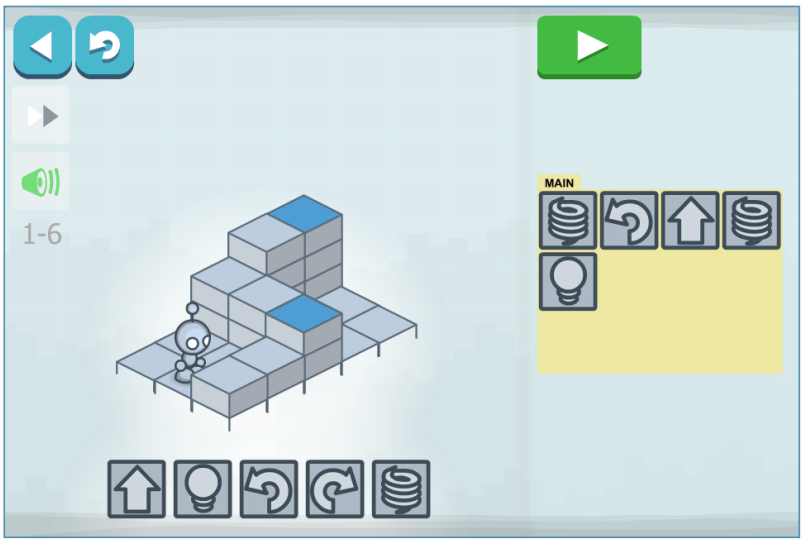
Du kan spille det gjennom internetbrowseren din ved å gå til denne linken:

`http://lightbot.com/hocflash.html`

Spillet er også tilgjengelig til smarttelefoner og tablets, både som en gratisapp og en lengre versjon som koster litt penger. Bare søk etter 'Lightbot' i din appstore, og last det ned.

Hvordan spiller du?

Under er et bilde fra spillet. Målet er å programmere den lille roboten sånn at alle de blå feltene på brettet lyser opp. De små ikonene på bunn av skjermen er de tilgjengelige kommandoene eller instruksene du kan bruke. På høyre siden av skjermen, i boksen som heter 'main', er selve programmet du lager. Du legger til instrukser i programmet ved å trykke på ikonene på bunn av skjermen. Hvis du er misfornøyd med det du har laget så langt kan du fjerne kommandoer fra programmet ved å trykke på dem. Du kan og enkelt bytte på rekkefølgen av instruksene ved å dra dem frem og tilbake.



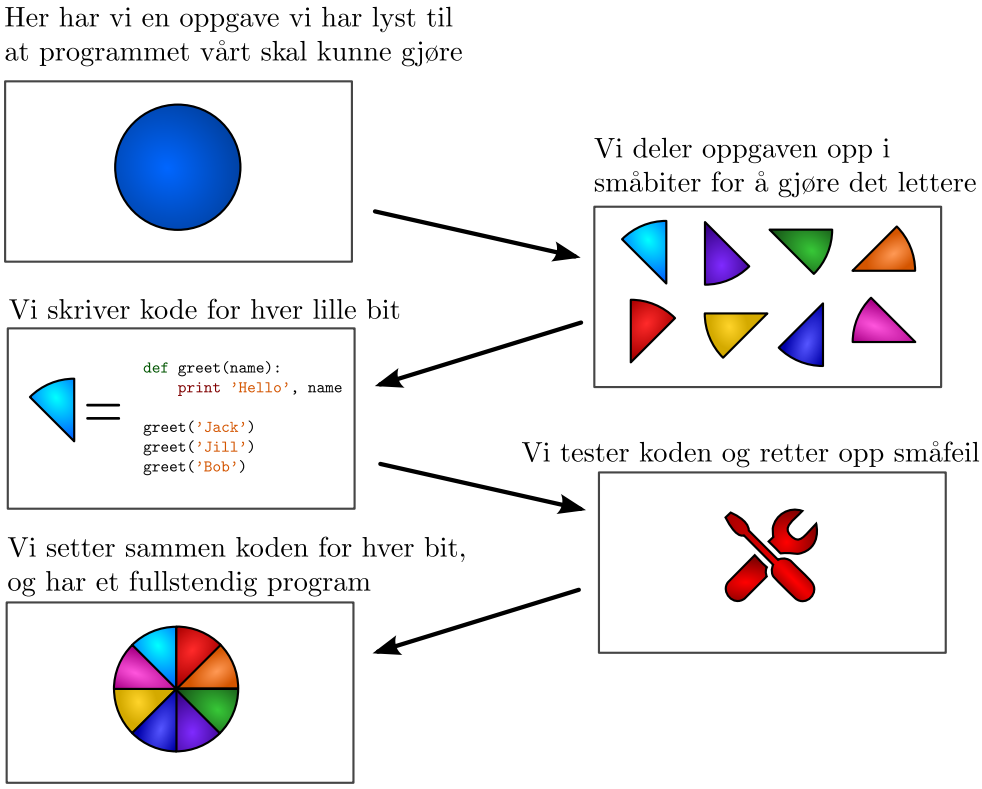
Når du er fornøyd med programmet du har laget til roboten trykker du på den grønne pilen på toppen av skjermen, da kjøres programmet. At programmet ditt kjøres, betyr at alle kommandoene du har lagt inn utføres i rekkefølge. Du kan nå følge med på hva roboten gjør, for å se om programmet ditt gjør som du ville. Enten greier roboten å lyse opp alle de blå feltene, isåfall har du greid brettet. Eller så mislykkes roboten, det gjør ingenting, da trykker du bare på den oransje knappen for å resette roboten, og du kan så gjøre endringer til programmet før du prøver å kjøre det igjen.

Etter du har spillt igjennom de fire, fem første brettene, begynner du nok å få teken på hvordan du skal programmere roboten til å gjøre det du vil. Det du gjør når du spiller dette spillet, er faktisk programmering. For å være mer nøyaktig, så er spillet et eksempelt på noe vi kaller *symbolsk programmering*, fordi du bruker symboler, eller ikoner, for å lage programmet ditt, istedenfor kode. Men ellers så er dette akkurat det samme som vi kommer til å gjøre når du skal lære å skrive Python.

For hvert brett så har du en konkret oppgave du har lyst til å løse, du har lyst til å lyse opp alle de blå feltene på brettet. For å klare det er det en del hindre du må komme deg forbi. Du starter å løse hvert brett ved å finne ut hvilke kommandoer du må bruke for å komme deg forbi hindrene og lyse opp feltene. Etter du har skjønnt hvordan du skal løse brettet, så 'koder' du opp programmet ditt ved å velge de riktige instruksene. Når du har gjort dette, så tester du koden din ved å kjøre programmet. Hvis det er feil i koden, så retter du dem opp. Slik fortsetter du, helt til du har et helt program som løser hele brettet. Vi har nå oppsummert hva det vil si å programmere.

Oppsummering av programmeringstankegangen

For å oppsummere programmeringstankegangen, så har jeg laget en liten figur. Ta en titt på den og se om du greier å knytte opp spillet Lightbot til de forskjellige stegene.

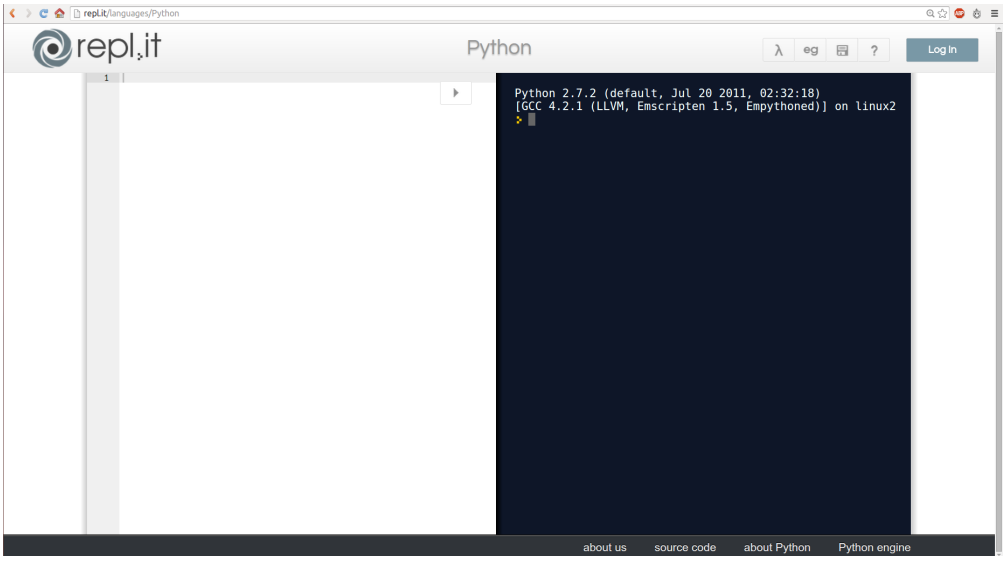


Pythonprogrammering

Nå som vi har fått inn grunntanken med programmering, la oss begynne å se på Python. Når dere skal programmere i Python så trenger dere en data hvor python er installert. Akkurat nå tar vi oss ikke tid til å installere det vi trenger, så vi velger å bruke en løsning på internet. Hvis vi går inn på nettsiden

`repl.it`

så kan du velge mellom mange forskjellige programmeringsspråk. Klikk på Python, og vi kommer inn i noe vi kaller en Python interpreter, der vi kan skrive og kjøre Python-kode. Nettsiden skal nå se ligne på dette



På venstre side av skjermen, i det hvite vinduet, er der vi skriver selve programmet vårt, vi kaller gjerne dette vinduet for editoren. I editoren skriver du koden din. På høyre side av skjermen, i det mørke-blå vinduet, er det resultatet av kjøringen din kommer. Vi kaller det blå vinduet for terminalen. Når du vil kjøre programmet du har skrevet i det hvite vinduet, så kan du trykke på pilen du ser øverst til høyre i det hvite vinduet (ca midt på skjermen), da kjøres programmet ditt. Når programmet kjøres, så utfører Python koden du har skrevet, linje for linje, fra toppen ned. Resultater fra kjøringen skrives ut i terminalen.

Ditt første Pythonprogram

Nå er det på tide at du skal skrive ditt første Pythonprogram. Vi lar det være et utrolig enkelt program, kun bestående av linjen

```
print 'Hello, world!'
```

Skriv inn koden i editoren (det hvite vinduet), og trykk på pilen for å kjøre programmet ditt. Hva er det som skjer? Forhåpentligvis så vises skriften *Hello, world!* i terminalen (det blå vinduet). La oss ta en titt på hva som har skjedd. Kodelinjen du har skrevet er en `print`-kommando. I Python så er `print` et nøkkelord som betyr at noe skal vises (skrives ut) til terminalen. Det koden vår gjør er altså å bruke `print`-kommandoen til å skrive ut en beskjed til terminalen.

Merk at i koden så har vi omringet beskjeden vi har lyst til å skrive ut med fnutter, `''`, dette er fordi Python skal kunne skille mellom teksten i beskjeden og resten av koden. Vi kaller teksten i fnutter for en tekststreng, og tekststrenger tolkes aldri av Python som kode, den behandles bare som en tekst. Hvis du ser på terminalen, så ser du at Python ikke har inkludert fnuttene i det den har skrevet ut, det er det som er mellom fnuttene som er viktig. Merk at hele tekststrenger har blitt farget en egen farge i editoren, sånn at det er lett å skille mellom teksten som er kode, og teksten som ikke er kode. Så lenge vi skriver beskjeden vi vil skrive ut som en tekststreng, det vil si mellom de to fnuttene, så kan vi skrive hvilken som helst tekst vi ønsker. Prøv selv å endre beskjeden et par ganger og se at alt fungerer som det skal.

La oss nå prøve å ha to printkommandoer etter hverandre

```
print "Hva skjera Baghera?"
print "Ingenting tingeling!"
```

Hva skjer når du kjører programmet nå? Hver av linjene i programmet er en `print`-kommando som skriver ut en tekststreng. Python tolker koden linje for linje nedover, så den vil først behandle den ene print-kommandoen, og så den andre. Utskriftene til Terminalen havner etter hverandre, på hver sin linje.

Variable

Når vi skriver et dataprogram, så er det viktig at datamaskinen skal kunne huske på ting den trenger å vite. Vi får datamaskinen til å huske på ting ved å definere noe vi kaller variabler. Vi kan tenke på en variabel som en boks. La meg vise deg et eksempel, her er to linjer kode

```
name = "Jonas"
alder = 23
```

Hva er det denne koden gjør? Den første kodelinjen sier `name = Jonas`. Det jeg gjør med denne kommandoen er å fortelle Python navnet mitt. Mer teknisk kan jeg si at jeg lager en variabel som heter `name`, og den inneholder navnet mitt, som er teksten 'Jonas'. Det Python gjør når den behandler denne kodelinjen, er at den oppretter en variabel, tenk på det som en tom boks, som heter `name`. Så putten den teksten 'Jonas' inn i den boksen, så setter den boksen i arkivet. Tilsvarende sier den neste kodelinjen at det skal være en variabel som heter `age`, og der ligger tallet 23.

Prøv å skriv et lignende program for deg selv, med ditt navn og din alder. Kjør programmet, hva skjer? Ingenting skjer, ihvertfall ikke som du kan se. Terminalen er helt tom. Python oppretter variablene, akkurat slik vi spør om, men den viser ikke i terminalen at den har gjort noe. For å få noe til å skje, så kan vi legge til en `print`-kommando i bunn av programmet vårt

```
print name
```

Hvis du kjører programmet nå, så ser du at navnet ditt skrives ut i terminalen. Det er fordi du ber Python skrive ut `name`, det som skjer da er at Python sjekker om den har en variabel (en boks) som heter name, når den finner den riktige variabelen i minnet sitt (den riktige boksen fra arkivet), så sjekker den hva

innholdet er, og skriver ut innholdet til terminalen. På samme måte kan vi skrive ut variabelen `age` til terminalen med kommandoen

```
print age
```

Det du kan prøve på nå, er å skrive

```
print 'name'
```

eller

```
print 'age'
```

Hvis du gjør det, så ser du at du *ikke* får skrevet ut navnet eller alderen din til terminalen. Det er fordi Python tolker det du prøver å skrive ut som tekststrenger, fordi du har skrevet dem med fnutter på sidene.

Merk at hvis du prøver å opprette to variabler med samme name, så overskrives rett og slett den originale variabelen. Prøv for eksempel programmet

```
name = "Marius"
name = "Lise"

print name
```

Hva forventer du at skjer? Prøv selv!

Typer

Du har nå sett at vi kan lage variable. Variabler har altså navn og innhold, men de har også en *type*. Hvis vi ser på de to variablene vi nettopp lagde, så har vi `name`, som inneholder en tekststreng, og `alder` som inneholder et tall. Når Python oppretter en variabel, så husker den ikke bare på navnet og innholdet, men den merker seg også hva slags innhold variabelen har.

Vi kan bruke kommandoen `type` til å sjekke hva slags type en variabel har. Her er et eksempelprogram

```
location = "Oslo"
year = 2015
day = "4. januar"
temperature = -7.3

print type(location)
print type(year)
print type(day)
print type(temperature)
```

I dette programmet oppretter jeg først fire variable, så skriver jeg ut til terminalen hvilken type de har. Merk at kommandoen `type` sjekker typen til det du skriver inne i parantesen, sånn at `type(sted)` betyr typen til variabelen `navn`. Når du kjører programmet får du denne utskriften i terminalen

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

Vi ser altså at variabelen `location` har type 'str', dette er en forkortelse for 'string', og betyr altså tekststreng. Dette stemmer jo godt, fordi variabelen `location` inneholder jo en tekststreng. Variabelen `year` har type 'int' som er en forkortelse for 'integer'. Integer er engelsk og betyr heltall. Vi ser at `day` også er en 'str', altså tekstreng. Til sist har vi `temperature` som har typen 'float', som betyr flyttall. Et flyttall er egentlig bare et annet navn på et desimaltall. Vi ser altså at Python skiller på heltall og desimaltall.

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.

Feilmeldinger

Nå som du har begynnt å skrive din første Pythonkode kan det jo være at det har oppstått noen feil, og hvis du har gjort alt rett så langt, så dukker det opp noen feil snart. La oss ta en titt på feilmeldingene vi får når vi gjør feil i Python, og prøve å tolke dem litt. Når du programmerer kommer du til å gjøre mange feil og det er viktig å prøve å forstå hvorfor det gikk galt, det å tolke sine egne feilmeldinger er nok den aller beste måten å bli god til å programmere.

La oss skrive en `print`-kommando feil med vilje

```
prnt "Hello, World!"
```

Når du prøver å kjøre programmet nå, så får du en feilmelding ut. Den ser ut noe som dette:

```
File "<stdin>", line 1
    prnt "Hello, World!"
                       ^
SyntaxError: invalid syntax
```

Det er alltid den nederste linjen i en feilmelding som er den viktigste, og der står det `SyntaxError: invalid syntax`. Feilen vi har gjort er altså en *syntaks*-feil. En syntaks-feil betyr at Python ikke skjønner det vi har skrever, vi har skrevet noe som rett og slett ikke gir mening. Når du får en syntaks-feil bør du altså sjekke at du har skrevet alt riktig, sånn som her ser vi at `print`-kommandoen er skrevet feil.

På linjene over prøver Python å informere oss hvor feilen er. Det står 'line 1' på toppen, akkurat nå er det jo åpenbart at feilen må være i kodelinje 1, for det er alt vi har skrevet! Men i et program på flere hundre kodelinjer er det veldig nyttig å få vite hvilken linje feilen er på.

La oss prøve en annen feil

```
location = "Oslo"
print place
```

Hvis du kjører dette programmet får du feilmeldingen

```
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
NameError: name 'place' is not defined
```

Nå har du ikke lenger en syntaks-feil, fordi Python skjønner godt hva du har lyst til å gjøre her, det du har skrevet er helt riktig Python-kode. Problemet er derimot en `NameError`, det er ganske enkelt fordi programmet først oppretter variabelen 'location', og prøver deretter å printe ut variabelen 'place'. Men det finnes ingen variabel med navn 'place', så derfor får du en navn-feil—programmet prøver å bruke en variabel som ikke finnes.

La oss se på en siste feil

```
students = ["John", "Jake", "Mary", "Marcus"]
print students[4]
```

Når programmet kjøres får du feilmeldingen

```
File "<stdin>", line 2, in <module>
    print students[4]
IndexError: list index out of range
```

Vi ser at vi får en `IndexError` og det står 'list index out of range'. Målet med `print` kommandoen er å skrive ut det fjerde navnet i lista, Marcus. Problemet er derimot at vi har glemt at Python starter å telle på 0, så Marcus har indeks 3! Dermed prøver vi å lese en del av lista som ikke finnes, og vi får en 'index out of range'-feil.

Mer om printing

Så langt har du sett at du kan skrive ut både tekststrenger og variable, la oss nå kombinere dem. Ta en titt på følgende program

```
name = "Silje"
print "Hei", name, "! Hvordan har du det idag?"
```

Her bruker vi `print`-kommandoen til å skrive ut 3 ting etterhverandre. Hvis du kjører programmet så ser du at de tre tingene vi skriver ut havner på samme linje, ikke hver sin linje, det er fordi de alle hører til samme `print`-kommando.

Hvis du ser nærmere på utskriften, så kan du legge merke til at Python har lagt inn et mellomrom mellom hver av tingene vi skriver ut, så i terminalen står det "Hei Silje ! Hvordan har du det idag?". Det er jo litt dumt at det er et ekstra mellomrom mellom 'Silje' og '!', så la oss se på en annen måte vi kan skrive ut en beskjed og en variabel samtidig.

```
name = "Silje"
print "Hei %s! Hvordan har du det idag?" % name
```

Prøv å kjør dette programmet? Nå ble beskjeden akkurat slik vi ønsket. Men hva er det vi egentlig har gjort her? Vi ser at vi prøver å skrive ut en tekststreng, men inne i tekstrengen står det `%s`. Når vi skriver `%s` inne i en tekststreng, så lager vi et slags hull i strengen, der vi kan fylle inn en variabel. Vi skriver `% name` bak tekstrengen, fordi det er denne variabelen vi ønsker å fylle inn i hullet. Grunnen til at vi skriver `%s` i strengen er fordi `s` står for streng, siden vi fyller inn med en variabel som er en tekststreng.

Vi kan lage så mange hull i en tekst som vi ønsker, her er et eksempel

```
name = "Silje"
age = 18
location = "Drammen"

print "Jeg heter %s, er %i og kommer fra %s." % (name, age, location)
```

Nå ser vi at det er 3 hull i teksten, det er to strenger merket med `%s` og et heltall merket med `%i` (husk at integer betyr heltall på engelsk). Bak strengen har vi listet opp tre variable som skal fylles inn i teksten. Merk at vi har satt parantes rundt dem, og at de kommer i samme rekkefølge som hullene i teksten.

Dataprogrammer som snakker med brukeren

Så lang har vi bare laget programmer som gjør noe enkelt og slutter helt av seg selv. Men de fleste programmer dere bruker i hverdagen er jo laget for å ha en interaksjon med brukeren. La oss derfor stille brukeren av programmet noen spørsmål, det kan vi gjøre med kommandoen `raw_input`. Her er et eksempel

```
weather = raw_input('Hi! How is the weather today?')
print "The weather seems to be %s today!" % weather
```

Når Python utfører denne kodelinja, så skrives spørsmålet i parantesen ut i terminalen, og så venter den på at brukeren som har kjørt programmet skal skrive inn et svar. Prøv å kjør programmet, skriv inn et svar på spørsmålet og trykk enter for å fortsette videre i programmet. Det du (brukeren) svarer

på spørsmålet blir så lagret i variablen `weather`. Etter du har trykket enter, fortsetter programmet. I vårt tilfelle går den da videre til å skrive ut en beskjed, hvor svaret brukeren ga brukes.

Oppsummering uke 1

- Å programmere betyr å lage dataprogrammer. Dette gjør vi ved å skrive kode i et bestemt programmeringsspråk. Koden er instrukser til datamaskinen. Før vi kan gå igang med å kode må vi ofte bryte ned oppgaven vi skal løse i små biter.
- Datamaskinen husker ting i form av *variabler*, vi kan opprette variabler ved å gi dem navn og et innhold. Python gir også variabelen en type, som vi kan sjekke ved å bruke kommandoen `type()`.
- Vi kan lage lister ved hjelp av firkantparanteser: `[, ]`. Lister er variabler som inneholder flere ting, de kan inneholde så mange elementer vi vil, av alle slag.
- Vi kan finne lengden på en liste ved å bruke `len()`. Vi kan også legge til fler elementer med `my_list.append()`. Enkelte elementer kan leses ut eller endres ved indeksering: `my_list[2]`. Husk at indekseringen begynner på 0. Så første element er `my_list[0]`, andre element er `my_list[1]` og så videre.
- For å skrive ut noe til terminalen kan vi bruke `print`-kommandoen, vi kan skrive ut både variabler og tekststrenger.
- Vi kan skrive ut flere ting etterhverandre hvis vi skiller dem med komma: `print ting1, ting2`, eller vi kan skrive ut tekster som vi fyller inn med variabler: `print "Hei, jeg heter %s" % name`.
- Vi kan spørre brukeren et spørsmål med kommandoen `raw_input()`, i parantesen skriver du spørsmålet som en tekststreng.
- Det er lett å gjøre feil i programmering, men det gjør ingenting. Når vi kjører et program med feil i, får vi en feilmelding som prøver å fortelle oss hva som har gått galt.

Oppgaver til uke 1

Oppgave 1 — Printing

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

(b) Lag et program hvor du lagrer navnet ditt som en variabel, og deretter skriver ut lengden på navnet ditt med kommandoen `len(name)`.

(c) Lag et program som spør brukeren om navnet dems, og deretter skriv en beskjed tilbake som bruker navnet de har gitt.
Hint: Du kan spørre brukeren et spørsmål med `raw_input`-kommandoen.

Oppgave 2 — Lister

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.

Oppgave 3 — Adjektivhistorie

(a) Spør brukeren om 5 adjektiv og lagre dem i 5 forskjellige variable.
Hint: Adjektiv er beskrivende ord som for eksempel 'liten' og 'stor'.

(b) Test programmet ditt ved å skrive ut alle 5 adjektivene i terminalen.

(c) Skriv en adjektivhistorie og print den slik at de 5 adjektivene brukeren har gitt fylles inn i historien.

(d) Test programmet ditt.

(e) Hvis alt funker som det skal kan du nå gjemme editoren din, kjør programmet og få en venn til å fylle ut adjektivene.

Oppgave 4 — Lightbot

(a) Spill igjennom alle brettene.

(b) **Utfordring**: Klarer du å løse brett 2–1 med bare 7 kommandoer? Hva med brett 2–2?

(c) **Utfordring**: Hvor få kommandoer klarer du å bruke for å løse 2-6? Det beste jeg har fått til er 15 kommandoer!

Matematikk i Python

Programmering og matematikk er tett knyttet sammen,

Du kan bruke vanlige matematiske operasjoner som `+` og `-` i Python, for å gjøre utregninger. Vi kan for eksempel skrive

```
print 12 + 49
print 24.4 - 6.3
```

Husk at vi fortsatt må bruke `print`-kommandoen for å faktisk skrive ut resultatet til terminalen. Hvis du bare skriver `2+2` på en kodelinje, så vil Python regne ut svaret, men så vil den ikke bruke det svaret til noe så utregningen er ganske bortkastet.

Du kan også lagre resultatet fra en utregning i en variabel

```
addition = 72 + 23
subtraction = 108 - 204
multiplication = 108 * 0.5
division = 108/9
```

I hvert tilfelle her så regner Python først ut det som er til høyre for likhetstegne, og lagrer resultatet i variabelen som står til ventre for likhetstegnet. Merk også at `*` er multiplikasjon og `/` divisjon. Et problem med divisjon i Python er at det finnes noe som heter heltallsdivisjon. Når du gjør beregningen `108/9`, så ser Python at du prøver å dele et heltall på et heltall (husk at Python skiller på heltall og desmialtall), siden du deler heltall på hverandre tror Python at du vil ha et heltall som resultat! Prøv for eksempel å skrive

```
print 5/2
```

Svaret du får er 2, ikke 2.5. Det er fordi vi har heltalldivisjon, og heltalldivisjon forteller deg bare hvor mange ganger telleren (her 5) går opp i nevneren (2), siden 5 går opp i 2 to ganger, får vi svaret 2. Hvordan unngår du heltallsdivisjon? Hvis et av tallene som inngår i divisjonen er et flyttall, så vil Python skjønne at du har lyst på flyttallsdivisjon (det du kjenner som 'vanlig' divisjon). Vi har altså et par muligheter for å få til dette

```
print 5.0/2.0
print 5.0/2
print 5/2.0
print 5./2
print float(5)/2
```

Alle disse måtene å dele tallene på hverandre vil gi svaret 2.5. Alle måtene fungerer fordi minst ett av tallene som inngår i divisjonen er flyttall, i det første tilfellet skriver vi `5.0` og `2.0` istedenfor `5` og `2`, men vi ser at det også går å bare skrive for eksempel `5.`. I det siste tilfellet så endrer vi typen til tallet `5` direkte med kommandoen `float`.

Du kan kombinere så mange matematiske operasjoner som du vil

```
print 305/2.0 + 222*5 - 3
```

Python utfører operasjonene i riktig matematisk rekkefølge, multiplikasjon og divisjon før addisjon og subtraksjon osv. Hvis du for eksempel skriver `1+2*3` så blir resultatet 7, *ikke* 9. Hvis du vil at utregningen skal skje i motsatt rekkefølge

kan du bruke paranteser akkurat sånn som man gjør for hånd: `(1+2)*3` gir resultatet 9.

Du kan også regne med variabler, så du kan for eksempel skrive

```
a = 3.0
b = 4.0
print 4*a
print a + b
print a * (b+2)
```

En ting det er verdt å være obs på, er at likhetstegnet i Python: `=`, ikke er det samme som i matte. Hvis du for eksempel skriver

```
x = 1
x = x + 2
```

Så er dette helt riktig Python kode som gir mening. I matematikk derimot, så gir ikke ligningen

$$x = x + 2,$$

noe mening i det hele tatt. Hvordan kan noe være lik seg selv *pluss to*?

Du må huske at likhetstegnet i Python kun brukes for å definere variable. La oss se på hvordan Python tolker kodelinjen

```
x = x + 2
```

Først ser den på høyre side av likhetstegnet og regner ut det som står der. Vi har definert variabelen `x` til å være 1, så på høyre side står det `1+2` som gir resultatet 3. Deretter ser den på venstre side av likehetstegnet, der skal den lagre resultatet. Siden `x` allerede er definert i programmet, så blir den rett og slett overskrevet av den nye kommandoen. Etter programmet har kjørt, så finnes det altså 1 variabel, `x` som har verdien 3. Du kan sjekke det selv ved å skrive ut resultatet med `print x`.

Hvis du syns likhetstegnet er litt vansklig å forstå, så kan det kanskje hjelpe å tolke det som en slags pil istedet, som egentlig gir mer mening

```
x <- 1
x <- x + 2
```

Dette er *ikke* gyldig kode, det er bare for å forklare hvordan ting fungerer i bakgrunnen.

Eksempel: Restaurantregning

La oss nå se på et eksempel hvor vi regner litt. La oss si vi er på restaurant, vi har spist og drukket for et grunnbeløp, men i tilegg til det kommer merverdiavgift (mva) og tips. La oss lage et program hvor vi først definerer grunnbeløpet, hvor mye mva vi må betale, og hvor mye tips vi har lyst til å gi. Deretter regner vi ut totalen vi må ut med og skriver den til terminalen.

Først definerer vi grunnbeløpet, mva, og tips.

```
basispris = 240
mva = 0.25
tips = 0.1
```

Her sier vi at vi grunnbeløpet er 240 kr, vi må betale 25 % mva, som er standarden i Norge, og vi har lyst til å tipse 10 %. Merk at jeg har skrevet prosentene som desimaltall ved å dele på 100, det er så det skal være lettere å regne med.

Nå skal vi regne ut totalen, først legger vi inn mvaen, deretter tipsen

```
pris = basispris + basispris*mva
tips = pris * tips
total = pris + tips
```

Til slutt kan vi skrive ut resultatet med en liten beskjed

```
print "Totalprisen ble %.2f kr, hvorav %.2f er tips." % (total, tips)
```

Når jeg skriver ut bruker jeg `%.2f`, `f` fordi det er flyttall jeg skriver ut, `.2` betyr at jeg bare vil ha med to desimaler i svaret som skrives ut. Prøv selv med bare `%f` og se hva som skjer, du får fryktlig mange desimaler.

Mer avansert matematikk

Du kan regne med potenser ved å bruke to produkttegn på rad `**`. For eksempel vil

```
print 4**2
print 2**3
```

gi resultatetene 16 og 8 fordi $4^2 = 16$ og $2^3 = 8$.

Hvis vi har lyst til å ta kvadratroten av et tall, for eksempel $\sqrt{49}$, så finnes det en kommando som gjør dette, men den må vi først *importere*. Grunnleggende Python er et ganske lite og enkelt programmeringsspråk, men det finnes tusenvis av pakker man kan inkludere for å gjøre det mer komplekst og kraftig. Nå skal vi drive med matte, og importerer derfor en kommando fra biblioteket som heter 'math', det gjør vi som følger

```
from math import sqrt
```

Vi har nå importert `sqrt` (står for square root, eller kvadratrot på norsk) fra 'math'. Vi kan nå bruke `sqrt` som alle andre kommandoer i Python, for eksempel til å regne ut $\sqrt{49}$:

```
print sqrt(49)
```

Alle kan lage ekstrabiblioteker til Python og det finnes derfor mange tusen biblioteker vi kan laste ned og bruke fritt når vi koder. Det finnes pakker for å lage grafikk, musikk, jobbe med nettsider osv. Hvis vi for eksempel ønsker å bruke Python til å lage et spill eller en iPhone app, så finnes det altså pakker til Python som gjør den jobben mye lettere for oss.

Nå tilbake til pakken 'math'. Vi importerte kvadratrotfunksjonen derfra. En annen ting vi kan importere er tallet π. Prøv å skrive

```
from math import pi
print pi
```

Nå har vi ikke importert en ny kommando fra math, men istedet en variabel, som inneholder tallet π til mange, mange desimaler.

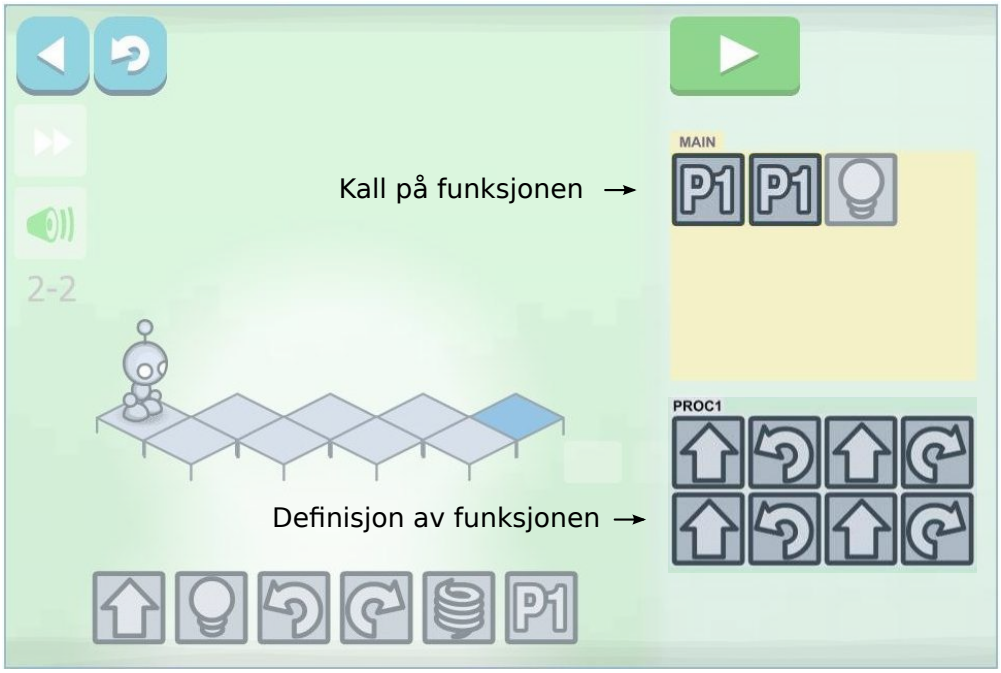
I math finnes det mange andre funksjoner og konstanter vi kan importere, men det er nok ikke så mange du har bruk for akkurat nå. Etterhvert som du lærer mer matematikk i for eksempel 1T matte, så kommer du til å lære om matematiske funksjoner som eksponentialfunksjonen, sinus, cosinus, tangens og fakultet. Mattebibloteket inneholder alle disse funksjonene og er derfor veldig nyttig når man skal drive med matte.

Funksjoner

Vi skal nå begynne å snakke om funksjoner. Funksjoner er noe av det mest fundamentale og viktige i programmering. Funksjoner lar oss definere egne kommandoer, og på den måten kan vi effektivisere koden vår enormt mye.

Du har allerede sett eksempler på funksjoner når du spilte Lightbot, der ble de kalt procedures. Når vi har en del kode som gjentas i et program, så er det mer effektivt å definere dem som en funksjon. Når vi har definert en funksjon, så kan vi bruke den ved å *kalle på den*.

I bildet under, så er det akkurat det vi gjør. I tillegg til hovedprogrammet vårt, så har vi et eget vindu hvor vi definerer funksjonen `P1`, i hovedprogrammet så kan vi da kalle på `P1`, da utføres alle kommandoene som ligger i definisjonen av funksjonen.



I Python kan du definere funksjoner med nøkkelorder `def`, en forkortelse for 'define'. Vi gir funksjonen vårt et navn og skriver all koden som inngår i funksjonen med et innrykk, la oss se på et eksempel

```
def greet():
    print "Hello, world!"

greet()
```

I denne koden ser du at kodeordet `def` brukes for å starte definisjonen av en funksjon, navnet jeg gir funksjonen min er `greet`, parantesene må være der, og vi kommer tilbake til hvorfor. Etter den første linjen, så kommer selve koden i funksjon. All koden som inngår i funksjonen må få et innrykk, eller indentering. Du kan lage et slikt innrykk med tab-knappen (du finner den rett over Caps Lock knappen, på venstre side av tastaturet). I dette tilfellet er det bare en enkelt linje som har fått innrykk, sånn at funksjonen vår består bare av 1 kodelinje.

På bunn av koden så kaller vi på funksjonen ved å skrive navnet på funksjonen med paranteser. Når vi gjør det så kjører Python koden som er inne i funksjonen. Merk at vi kan kalle på funksjonen så mange ganger vi vil, hvis vi skriver

```
greet()
greet()
greet()
```

så skrives "Hello, World!" ut tre ganger i terminalen.

Så langt er funksjoner i Python helt like de du så i Lightbot, men nå skal vi innføre en forskjell som gjør funksjoner i Python mye bedre, *input*. Det er kanskje lettest å vise med et eksempel

```
def greet(name):
    print "Hi there %s, nice to meet you!" % name

greet("John")
greet("Mary")
```

Funksjonen heter det samme som før, men nå står det ikke bare tomme paranteser, det står `(name)`, dette betyr at funksjonen forventer at brukeren skal gi funksjonen noe inn når man kaller på den. Slik input kalles for *argumenter*, vår funksjon forventer å få et navn som argument. Når du ser på koden inne i funksjonen skjønner du kanskje hvorfor, det funksjonen gjør er å skrive ut en beskjed som inneholder navnet man bruker til å kalle på funksjonen. Når du kaller på en funksjon som skal ha argumenter, så skriver du dem inn i parantesene når du kaller på funksjonen, slik som er vist i bunn av koden.

Funksjonene jeg har vist deg hitil, har alle skrevet noe ut til terminalen, men det vi vanligvis vil at en funksjon skal gjøre, er å *returnere* noe. At en funksjon returnerer noe vil rett og slett si at den gir noe tilbake, vi kan tenke på det som funksjonens *output*. Tenk en stund på kvadratrotfunksjonen vi importerte fra `math`-biblioteket. Vi importerer `sqrt`, og det er faktisk en funksjon! Tenk på når du bruker den, da må du enten skrive ut resultatet med `print`:

```
print sqrt(4)
```

eller du kan lagre resultatet i en variabel

```
result = sqrt(4)
```

Ta en titt på det siste tilfellet, her definerer vi en variabel som heter `result`. Som forklart tidligere, vil Python her først regne ut det som er på høyre side av likehetstegnet, og så lagre resultatet i variabelen på venstre side av likhetstegnet. Spørsmålet er da: hva er egentlig resultatet av et funksjonskall? La oss se på et eksempel hvor vi bruker `return` i funksjonen vår

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

Her definerer jeg en funksjon som heter `double`, den tar et tall `x` inn, og *returnerer* $2x$, altså det dobbelte. Jeg kan nå bruke `double` på alt av tall. Men her bør jeg passe meg litt. Hvis jeg bare skriver:

```
double(2)
```

Så skjer det ingenting, det er fordi funksjonen selv ikke printer noe, jeg må istedet skrive

```
print double(2)
```

eller

```
result = double(2)
print result
```

Når Python ser kommandoen `print double(2)`, så må den først finne ut hva den skal skrive ut til terminalen, så den kjører koden i `double`-funksjonen, der ganges $2 \cdot 2$ og vi får resultatet 4. Så returneres resultatet, altså tallet 4. Det at resultatet returneres betyr at Python skjønner at det er dette som skal skrives ut til terminalen.

Hvis vi istedet hadde laget double med en `print`, istedet for en `return`, sånn her:

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

Så kan vi skrive bare

```
double(2)
```

For å få skrevet ut resultatet til terminalen. Men nå vil dette gi et rart resultat

```
result = double(2)
print result
```

Hva er det egentlig som skjer her? Prøv å kjør programmet og se hva du får ut. Først får du skrevet ut tallet 4 til terminalen, dette skjer allerede i den første kodelinja når Python ser funksjonskallet til `double`. Men så skal Python lagre resultatet i result, men siden vi ikke har noen `return`-kommando i funksjonen vår, så vet ikke Python hva den skal lagre i `result`, så derfor er `result` rett og slett en helt tom variabel. Når vi da på neste linje prøver å skrive ut `result`, så står det bare `None`.

For å oppsummere. I Python så har alle funksjoner et navn, de kan ta null, én eller fler argumenter og gi null, en eller flere outputs. All kode som skal være en del av koden må få et innrykk og for å returnere noe fra funksjonen (output) bruker du `return`-kommandoen. Her er malen for en funksjon

```
def my_function(argument1, argument2)
    ...
    ...
    ...
    return output1, output2
```

Før vi går videre anbefaler jeg deg å tenke litt på hvorfor vi egentlig kaller det å returnere. Vel, en ting er jo at funksjonen gir noe tilbake, et resultat av noe slag, det er jo greit nok. Men det er også en annen grunn. Husk at jeg har forklart at Python alltid tolker koden din linje for linje, en kommando av gangen. Når du kaller på en funksjon, så utfører Python den koden som er definert inne i den funksjonen, så på en måte kan du tenke på det som at Python gjør et hopp i koden din, den hopper inn i funksjonskoden. Når du gir en `return`-kommando, så sier du at funksjonen er ferdig. Det betyr at Python kan gå tilbake til det der den var i den originale koden og fortsette å tolke neste linje med kode, altså at den kan returnere til der den var. Dette konseptet illustreres ganske godt i Lightbot, der hver kommando som utføres lyser opp i programmet ditt! Når du bruker en funksjon, så ser du at hver kommando i funksjonsboksen lyser opp og utføres før programmet 'returnerer' til hovedprogrammet og fortsetter.

Det at koden hopper tilbake til der den var betyr noe ganske viktig, og det er at all kode som kommer etter en `return`-kommando, aldri blir utførst. Ta en titt på dette programmet

```
def greet(name):
    return len(name)
    print "Hi there %s" % (name)
```

Denne funksjonen tar et navn inn som argument og prøver å gjøre to ting med det. Den vil både skrive ut en beskjed til brukeren rett til terminal, og den vil også returnere antall bokstaver i navnet. Prøv å kjør koden, beskjeden blir aldri skrevet ut—det er fordi `return`-kommandoen markerer slutten av koden for Python.

Funksjoner i matematikk

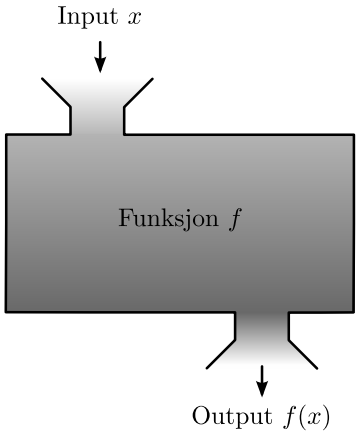
Jeg har nå vist deg litt hvordan du kan lage funksjoner i Python. Funksjoner er en av de viktigste konseptene i programmering, og nesten uansett hva slags program du skriver eller hvilket språk du bruker, kommer du til å bruke mye funksjoner. Men funksjoner er ikke bare viktig i programmering, det er også viktig i matematikk.

I matematikken kan du tenke på en funksjon som en slags regel som gjør om et tall, til et annet tall. Eller du kan tenke på det som en slags maskin, du mater et tall, gjerne kalt x inn en funksjon, ofte kalt f. Resultatet skriver vi $f(x)$, og du kan lese dette som 'resultatet når funksjonen f virker på tallet x'. Ofte så sier man bare 'f av x' for $f(x)$.

Et eksempel på en matematisk funksjon er for eksempel

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

Her definerer vi akkurat hva funksjonen vår gjør med et tall x. Vi kan nå sette



inn for hva som helst av tall vi vil for x, her er et par eksempler

$$f(0) = 0^2 + 2 \cdot 0 + 1 = 1,$$
$$f(1) = 1^2 + 2 \cdot 1 + 1 = 4,$$
$$f(2) = 2^2 + 2 \cdot 2 + 1 = 9,$$

Funksjoner er noe av det viktigste i matematikken, og det er på ungdomsskolen dere begynner å lære om dem. Når dere starter på videregående matematikk kommer det til å handle mye om funksjoner, og der lærer man om *derivasjon* og *integrasjon*, som handler mye om funksjoner. Hvis noen av dere en dag tar matematikk på et enda høyere nivå, kommer dere til å merke at det aller meste handler om funksjoner.

Jeg kommer ikke til å gå mer inn på matematiske funksjoner i dette kurset, siden vi fokuserer på programmering, og ikke nødvendigvis matematikk. Men matematiske funksjoner og funksjoner i programmering har veldig mye til felles, og det å ha forståelse for matematikk kommer til å hjelpe mye når man skal skrive kode og visa versa.

Oppsummering av uke 2

- Du kan bruke Python som en kalkulator med vanlige matematiske operasjoner: `+`, `-`, `*` og `/`. Du kan ta potens med `**`.
- Du må være forsiktig når du bruker divisjon for Python bruker *heltallsdivisjon*. For å unngå dette må du definere minst et av tallene som inngår i divisjonen som et desimaltall
- Python gjør kalkulasjoner i riktig matematisk rekkefølge. Du kan bruke paranteser for å bestemme rekkefølgen operasjonene gjøres i, akkurat som for hånd.
- Likhetstegn betyr *ikke* det samme i programmering som det gjør i matematikk. I programmering betyr likhetstegn bare at du oppretter en variabel, i matematikk betyr det rett og slett at to ting er helt like. For eksempel `x=x+2` fungerer fint i programmering, men ikke i matte.
- For mer avansert matematikk kan du importere ting fra et ekstrabibliotek. For eksempel kan du importere kvadratrot ved å skrive

```
from math import sqrt
```

  På samme måte kan du importere konstanter som π.

- Vi kan definere funksjoner i Python som fungerer som egne kommandoer. De kan ta argumenter inn, og kan returnere resultater. Etter vi har definert en funksjon, så kan vi kalle på den å mange ganger vi vil. Koden for å skrive en funksjon er strukturert som følger

```
def my_function(argument1, argument2)
    ...
    ...
    ...
    return output1, output2
```

Oppgaver til uke 2

Oppgave 1 — Kalkulator

Regn ut følgende ting og skriv resultatene til terminalen.

(a) $4 \cdot 5 + 2$

(b) 2^{10}

(c) $(-1) \cdot (-1)$

(d) $\sqrt{36}$ **Hint:** Du må importere `sqrt`-funksjonen

(e) $2 + 3 \cdot (4 - 1 + 7)$

(f) $\frac{1}{3} + \frac{2}{3}$

Syns du resultatene ser rimelige ut?

Oppgave 2 — Gjett på resultatet

Ta en titt på følgende program

```
y = 3
print y
y = y + 4
print y
y = y*y
print y
```

(a) Uten å faktisk kjøre programmet, kan du si hva som vil vises i terminalen når programmet kjøres?

(b) Etter du har prøvd å gjette på hva som vil vises i terminalen, kjør koden og se om du hadde rett.

Oppgave 3 — Finn feil!

Finn feilen i følgende program (det er én feil per program). Kjør gjerne programmet for å se feilmeldingen de produserer! Rett opp programmene så de fungerer som de skal.

(a)
```
name = Per
print "Hi there %s!" % name
```

(b)
```
x = 24
y = 36
print X + y
```

(c)
```
def greet(name)
    print "Hi there %s" % name
```

(d)
```
location = "Oslo"
temperature = -18
print "The temperature in %s is now %i" (location, temperature)
```

(e)
```
def double(x):
    result = 2*x

print double(2)
```

(f)
```
def greet(name):
print "Hi there %s" % name
greet("Mary")
```

Oppgave 4 — Kvadrattall

Skriv en funksjon som tar et tall som argument inn, og returnerer kvadrattallet.
Hint: Et kvadrattall er et tall ganget med seg selv, for eksmepel så er $3 \cdot 3 = 9$, så 9 er kvadrattallet til 3.

Oppgave 5 — Temperaturomregning

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 en funksjon som tar en temperatur i celsius som argument inn, og returnerer temperaturen i Fahrenheit.
Hint: Funksjonen din bør se noe ut som dette (fyll inn for prikkene)

```
    def celsius_to_fahrenheit(C):
        ...
        return F
```

Hint: Du kan kalle funksjonen din med for eksempel

```
    print celsius_to_fahrenheit(20)
```

(b) Sjekk at funksjonen din funker ved å vise at

- $0°$ Celsius svarer til $32°$ Fahrenheit
- $-40°$ Celsius svarer til $-40°$ Fahrenheit
- $100°$ Celsius svarer til $212°$ Fahrenheit

Programstruktur

Nå har dere jo begynnt å skrive en del kode. En ting som er viktig å huske på når man koder, er at man bør strukturere koden så den er så ryddig så mulig. All kode du skriver, bør være forståelig for andre som skulle lese den. Kode er jo egentlig skrevet for at en datamaskin skal forstå den, men det er også viktig at mennesker skjønner hva en kode gjør, ellers hjelper det ikke stort. Her er et populært sitat i programmeringsverden:

> *"Programs must be written for people to read, and only incidentally for machines to execute."*

Selv om du er sikker på at du aldri skal dele koden din med noen, er det lurt å gjøre den forståelig og ryddig. En ting er at du gjør det lettere for deg selv å rette opp feil i programmet ditt. En annen ting er at det blir lettere for deg å gå tilbake til koden din senere og endre på ting eller å videreutvikle ting.

Så det er altså lurt å strukture koden din, så det er lett å lese, men hvordan gjør du egentlig det? La oss ta en titt

Kommentarer

Det første du kan gjøre, er å skrive *kommentarer* i koden. Kommentarer er deler av programmet ditt som Python ikke tolker som kode, og som ikke påvirker programmet ditt på noen måte. Det eneste kommentarer gjør, er å forklare hva som skjer til de som leser koden din. Du skriver kommentarer med #-symbolet. Alt som står bak en # på en kode-linje tolkes som en kommentar, og vil ikke endre noe på programmet ditt. La oss se på noen eksempler

```
# Ask the user for his or her name
name = raw_input("Hi there, what's your name?")

# Greet the user with a nice message
print "Nice to meet you %s, I hope you have a great day!" % name
```

Når du kjører dette programmet, så fungerer det helt likt som hvis kommentarene ikke hadde vært der i det hele tatt, men de forklarer til en som leser koden hva som foregår.

Et vanlig sted å ha kommentarer, er i begynnelsen av funksjoner, for å forklare hva en funksjon gjør

```
def Fahrenheit2Celsius(F):
    # Converts a temperature from degrees Fahrenheit to degrees Celsius
    C = (5./9)*(F - 32)
    return C
```

Hvis du vil ha en kommentar som går over flere linjer, kan du skrive det hvis du bruker tre apostroffer på hver side av kommentaren: """. En ting du kan bruke det til, er å forklare på starten av et lengre program, hva programmet ditt gjør.

```
"""This is a comment
that covers
three lines in total"
```

Whitespace

En annen ting som er viktig for strukturen på et program, er det som kalles *whitespace*. Whitespace er ganske enkelt forklart alt vi ikke kan se, så det betyr mellomrom, tabs og tomme linjer. I kode er det noen steder det går fint å legge inn ekstra mellomrom, og andre steder det ikke går. Hvis du lærer deg hvor det er greit å bruke litt ekstra 'tomrom', så kan programmet ditt fort bli veldig mye finere.

Du kan for eksempel alltid legge inn blanke linjer i et program. Python bryr seg ikke om blanke linjer. Ved å lage litt pusterom mellom forskjellige deler av et program, så blir det for mye mer leselig. Se for eksempel på følgende programmer

```
from math import sqrt
number=raw_input("Please give me a number!")
root=sqrt(number)
print "The square root of your number is %d" % root
```

```
from math import sqrt

number = raw_input("Please give me a number!")
root = sqrt(number)

print "The square root of your number is %d" % root
```

De to programmene er akkurat samme kode, jeg har bare lagt inn noen ekstra mellomrom og blanke linjer i det siste programmet. Jeg syns ihvertfall det siste programmet er mye mer oversiktelig og leselig enn det første. I programmer med flere hundre kodelinjer er tomrom som dette utrolig viktig, så man får litt 'pusterom' når man leser koden.

Random

Mange dataprogrammer har elementer av tilfeldigheter innebygget. Et godt eksempel er spill, men det er også viktig i datasikkerhet og brukes mye i vitenskaplige simuleringer. Du skal nå få lære hvordan vi kan lage tilfeldigheter i programmet ditt.

Vi kommer til å bruke biblioteket `random` for å få det vi trenger av funkjsoner. Navnet 'Random' er engelsk og betyr 'tilfeldig'. Husk at du kan importere enkelte funksjoner fra et bibliotek med `import`. Sånn at hvis du vil for eksempel ha funksjonen `randint` (vi viser snart hva denne gjør) fra biblioteket `random` skriver du

```
from random import randint
```

Hvis du vet at du kommer til å bruke mange forskjellige funksjoner fra et biblotek, så kan du istedet skrive

```
from random import *
```

Stjernetegnet betyr at vi importerer alle funksjoner fra et bibliotek.

Rulle terning

La oss nå se på hva funksjonen `randint` faktisk gjør. Navnet på funksjonen `randint` er en sammentrekkning av 'random integer', den gir oss altså et tilfeldig heltall. Funksjonen tar to argumenter inn, to heltall a og b. Funksjonen returnerer et heltall fra og med a til og med b. La oss lage et enkelt program som ruller en vanlig seks-sidet terning

```
from random import randint

# Rolling a die
result = randint(1,6)
print result
```

Når programmet kjøres, så vil `randint` returnere et tall fra 1 til 6, og vi skriver ut resultatet. Hvis du kjører programmet mange ganger etterhverandre, så vil du se at du får et tilfeldig resultat hver gang.

Vi kan selfølgelig bruke andre argumenter inn til randint for å simulere alle slags forskjellige 'terninger'.

```
from random import randint

# 20-sided die
print randint(1,20)

# Coin-flip
print randint(0,1)

# Two six-sided dice
print randint(1,6) + randint(1,6)
```

I dette programmet bruker vi randint på tre forskjellige måter.

Et par andre funksjoner fra `random` er laget for å virke på lister. Vi har for eksempel `shuffle`, som rett og slett stokker om på elementene i en liste så de ligger i en tilfeldig rekkefølge.

```
from random import shuffle

numbers = [1, 2, 3, 4, 5]
shuffle(numbers)

print numbers
```

Mens funksjonen `choice` trekker et tilfeldig element fra en liste. Vi kan for eksempel bruke `choice` til å holde et lotteri

```
from random import choice

students = ["Lisa", "Marcus", "Jake", "Mary", "Molly", "Blake", "Kane"]
winner = choice(students)

print winner
```

Eksempel: Kortstokk

La oss se på hvordan vi kan bruke lister og shuffle til å lage en kortstokk som kan brukes til å lage kortspill. Hvis vi lar vært kort i en kortstokk være representert av en tekststreng på to bokstaver, der den første er fargen og den andre valøren.

Vi bruker engelske navn for enkelhetens skyld (på norsk bruker f.eks knekt og konge samme bokstav, osv). For eksempel skriver vi spar-6 som 's6' og kløver dame some 'cQ'. Vi kan da definere en kortstokk på følgende måte (merk at kortstokk heter 'deck' på engelsk)

```
from random import shuffle

# Define our complete deck of cards
deck=[
's2', 'c2', 'd2', 'h2', 's3', 'c3', 'd3', 'h3', 's4', 'c4', 'd4', 'h4',
's5', 'c5', 'd5', 'h5', 's6', 'c6', 'd6', 'h6', 's7', 'c7', 'd7', 'h7',
's8', 'c8', 'd8', 'h8', 's9', 'c9', 'd9', 'h9', 'sT', 'cT', 'dT', 'hT',
'sJ', 'cJ', 'dJ', 'hJ', 'sQ', 'cQ', 'dQ', 'hQ', 'sK', 'cK', 'dK', 'hK']

# Randomize the order of the cards
shuffle(deck)
```

Ved å behandle listen som representerer kortstokken kan vi dele ut kort til spillere, fjerne kort, legge til kort, stokke og lignende. Vi har altså tatt første steg mot å lage et kortspill! Her måtte vi skrive ut hele kortstokken selv, men vi skal senere se en lur måte vi kan spare en del jobb og la kortstokken lage seg selv.

La meg vise deg noen kjappe måter å manipulere kortstokken på. Hvis du har en liste, kan du bruke `.pop()` til å fjerne siste element i lista, dette blir altså som å trekke et kort fra kortstokken:

```
print len(deck)
print deck.pop()
print len(deck)
```

Så først har variabelen deck, som inneholder alle de 52 forskjellige kortene i en tilfeldig rekkefølge. Vi trekker så ut siste kortet i listen, skriver det til skjerm, og ser da at listen bare er 51 lang, det er fordi 'pop' faktisk fjerner kortet, akkurat som når du trekker et kort fra en kortstokk. La oss trekke en pokerhånd på 5 kort, vi lar hånda vår være representert som en egen liste

```
# Make an empty list for our hand, so we have somewhere to put our cards
hand = []

# Draw 5 cards from the deck and put them into our hand
hand.append(deck.pop())
hand.append(deck.pop())
hand.append(deck.pop())
hand.append(deck.pop())
hand.append(deck.pop())

# Look at our hand
print hand
```

Igjen finnes det flere måter å gjøre denne koden bedre og mer elegant på, som dere snart skal lære—men dette viser hvordan vi kan begynne å lage spill fra bunn av på datamaskinen.

Tester

Du har nå sett hvordan du kan få tilfeldige tall i programmet ditt, la oss nå se på hvordan du kan få programmet ditt til å gjøre forskjellige ting basert på resultatet. For å gjøre det trenger vi det vi kaller for 'if'-tester. Grunntanken med 'if'-tester, er at datamaskinen kan sjekke om noe stemmer eller ikke—vi kan spesifsere betingelser for at noe skal skje.

La oss starte med et eksempel

```
from random import randint

result = randint(1,6)

if result == 6:
    print "You won the prize! Congratulations!"
else:
    print "Sorry, you lost. Better luck next time!"
```

I dette programmet importerer vi først funksjonen `randint` fra `random`, og bruker den til å trekke et tilfeldig tall mellom 1 og 6. Så bruker vi en `if`-test, slik at programmet skriver ut en beskjed hvis resultatet ble 6, og en annen beskjed hvis resultatet ble 1 til 5. Hvis ser at programmet forgrener seg og gjør enten en ting, eller noe annet. Kodelinjen vi bruker for å få til dette er

```
if result == 6:
```

Nøkkelordet her er `if`. Ordet 'if' betyr jo 'hvis'. Så vi sier 'hvis resultatet er 6, gjør dette'. Merk at vi bruker to likhetstegn på rad, dette er fordi et enkelt likhetstegn brukes for å lage variabler i Python, så vi bruker to likhetstegn for å si at to ting er like. En annen måte vi kunne skrevet betingelsen på er

```
if result is 6:
```

Altså betyr `is` og `==` det samme. Vi ser at etter `if`-testen, så har `print`-kommandoen fått et innrykk, akkurat som når vi lager funksjoner. All kode som har fått et innrykk er en del av det vi kaller `if`-blokken. Koden i `if`-blokken kjøres kun hvis betingelsen stemmer. Hvis betingelsen *ikke* stemmer, så kjører vi istedet all kode som hører til `else`-blokken. 'Else' betyr ellers, altså er 'if-else' kode enten-eller. Enten gjør vi den ene tingen, eller så gjør vi den andre—vi gjør aldri begge deler.

En `if`-test skrives altså generelt

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

Du trenger ikke alltid en `else`-blokk, kanskje du vil skrive ut en beskjed til brukeren bare hvis man ruller 6. Da dropper du bare `else`-blokken.

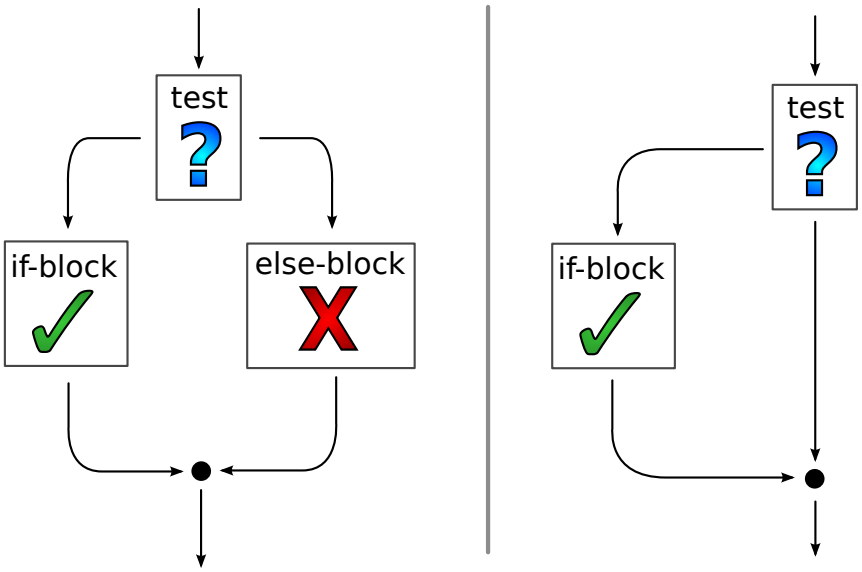


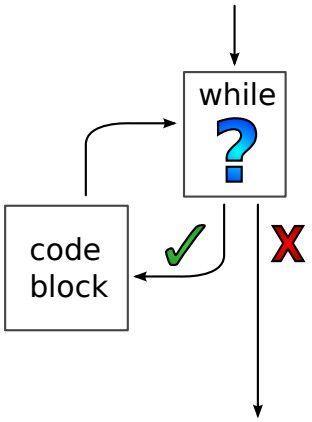
Figure 1: Her er en figur som viser programflyten i et program med en test. På venstre side ser du en test med både en `if`- og en `else`-blokk, mens til høyre er det bare en `if`-blokk.

Løkker

Løkker er det siste konseptet vi kommer til å gå igjennom i dette kurset. En løkke er en del kode som gjentar seg selv. Hvis vi vil at programmet vårt skal gjøre det samme mange ganger på rad, så må vi lage en løkke. Det finnes to typer løkker, vi kaller dem for `while`- og `for`-løkker. Vi komme til å se på begge to, men la oss starte med `while`-løkker.

While-løkker

En `while`-løkke ligner veldig på en `if`-test, fordi den sjekker en betingelse på samme måte som en `if`-test. Den store forskjellen er at en `while`-løkke vil gjenta seg selv helt til betingelsen blir falsk.



La oss se på et eksempel som viser forskjellen på en `if` og en `while`. La oss lage et spill hvor vi ruller en terning. Hvis du ruller en sekser, så vinner du, ellers taper du. Først lager vi spillet med en `if`-test.

```
from random import randint

# Roll a die
result = randint(1,6)
if result != 6:
    print "You rolled a %i, no prize for you this time." % result
else:
    print "You rolled a six! Great job!"
```

Når programmet kjøres, så trekker vi et tilfeldig tall mellom 1 og 6, så bruker vi en `if`-test til å skille på resultatet. Hvis resultatet er fra 1 til 5 så skrives beskjeden i `if`-blokken ut, og hvis resultatet er 6 så skrives beskjeden i `else`-blokken ut. Hver gang programmet kjøres, så vil det rulles et nytt tilfeldig resultat, men uansett hva som skjer, så rulles terningen bare 1 gang, og det skrives bare ut 1 beskjed.

Nå lager vi spillet med en `while`-løkke

```
from random import randint

# Roll a die
result = randint(1,6)

while result != 6:
    print "You rolled a %i, no prize for you this time." % result
    result = randint(1,6)

print "You rolled a six! Great job!"
```

Når dette programmet kjøres, så vil igjen et tilfeldig tall trekkes. Hvis tallet er 6, så er betingelsen "`result != 6'' i \verb`while+-løkka falsk, og vi hopper over all koden i løkken. Dermed går programmet rett til siste linje av programmet som skriver ut vinner-beskjeden. Hvis vi derimot ruller fra 1 til 5, så er betingelsen sann, og da kjøres koden i løkka. Vi skriver først ut taper-beskjeden, *og så ruller vi terningen på nytt.* Etter koden inne i løkka er skrevet ut, sjekkes betingelsen på nytt. Hvis vi nå rullet fra 1 til 5 på nytt, så er betingelsen sann og løkka kjøres på nytt. Sånn fortsetter det helt til vi ruller en 6'er og vinner. Her er et resultat jeg fikk når jeg kjørte på programmet

```
You rolled a 1, no prize for you this time.
You rolled a 4, no prize for you this time.
You rolled a 5, no prize for you this time.
You rolled a 5, no prize for you this time.
You rolled a six! Great job!
```

Her rullet jeg altså først 1, så 4, så 5, så 5 og til slutt 6. Merk at med `while`-testen vet vi ikke hvor mange ganger vi kommer til å kaste terningen, mens med `if`-testen så rullet vi terningen bare 1 gang, alltid.

La oss se på et nytt eksempel. Denne gangen kan vi se på hvor mange ganger vi må dobble et tall før det blir større en 1 million

```
i = 0 # number of doublings
n = 1

while n < 1000000:
    i = i + 1
    n = n*2
    print "After %i doublings, the number is: %i" % (i, n)
```

Og resultatet blir som følger

```
After 1 doublings, the number is: 2
After 2 doublings, the number is: 4
After 3 doublings, the number is: 8
After 4 doublings, the number is: 16
.
.
.
After 19 doublings, the number is: 524288
After 20 doublings, the number is: 1048576
```

Det som skjer i programmet er at `while`-løkka kjøres helt til tallet n blir større en 1 million, hver gang løkka kjøres, så blir n dobbelt så stor. Samtidig lar vi tallet i bli 1 en større hver gang løkka kjøres, det lar oss skrive ut til skjermen hvor mange ganger løkka har kjørt.

Uendelige løkker

Men en `while`-løkke er det veldig enkelt å lage en uendlig løkke. Tenk deg for eksempel programmet

```
from random import randint

result = randint(1,6)

while result != 6:
    print result
```

Her er idéen at vi skal rulle en terning helt til vi får 6. Problemet er at vi glemmer å rulle terningen på nytt inne i løkka! Hvis du kjører programmet nå, og for eksempel ruller 3, så vil programmet bare skrive ut resultatet til skjerm på nytt og på nytt uendelig lenge. Siden terningen aldri rulles på nytt, så vil betingelsen alltid være sann og programmet kommer aldri ut av løkka.

Hvis du lager et slikt program ved uhell, så må du avbryte det selv. Akkurat hvordan det gjøres avhenger av hvilken platform og hvilke verktøy du bruker. I Canopy, så kan du avbryte programmer ved å gå inn å toppmenyen "Run" og så klikke på "Interrupt Kernel" eller eventuelt "Restart Kernel". Hvis alt låser seg, så må du rett og slett restarte hele Canopy. Det er ganske slitsomt når ting låser seg, så du bør være forsiktig med å lage uendelige løkker!

For-løkker

En `for`-løkke gjentar også en bit med kode om og om igjen, akkurat som en `while`-løkke. Men istedenfor å gjøre det helt til en betingelse er usann, så må en `for`-løkke vite akkurat hvor mange ganger vi skal gjenta koden. En bedre måte å si det på er at `for`-løkke gjentar en bit med kode for hvert element i en liste. La oss for eksempel se på en liste med navn på elever i en klasse, vi kan da gjør for eksempel dette

```
students = ["Mary", "James", "Siri", "Alexander", "Elizabeth"]

for name in students:
    print "The name %s has %i characters" % (name, len(name))
```

Denne koden skriver ut en bestemt melding for hvert navn i lista, der meldingen avhenger av navnet. Utskriften vi får er som følger

```
The name Mary has 4 characters
The name James has 5 characters
The name Siri has 4 characters
The name Alexander has 9 characters
The name Elizabeth has 9 characters
```

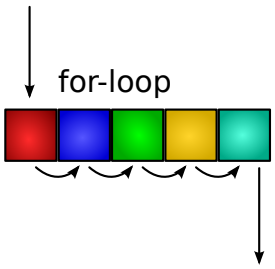
Så vi ser at koden i løkka blir kjørt for hvert navn i lista. Ettersom at students er en liste med 5 elementer, så kjøres koden i løkka 5 ganger.

En `for`-løkke begynnes alltid med nøkkelorder `for`. Det som står rett etter `for` er navnet på elementet vi behandler, i koden over er elementene i lista navn, og det er det vi skriver ut. Du kan selv velge hva du kaller elementet du behandler, og det kan være hva som helst. Etter navnet på det bestemte elementet kommer

nøkkelorder `in` så sluttes det helte med en liste og et kolon: `:`. Altså kan en `for`-løkke startes som dette

```
for my_element in some_list:
```

Koden som følger får et innrykk, akkurat som for tester og funksjoner. All kode som har et innrykk hører til løkken, og kommer til å bli gjentatt for hvert element i lista. Hver gang løkka kjøres, så vil innholdet i variabelen `my_element` være det neste elementet i lista, og koden som utføres vil derfor utføres på et nytt element hver gang. La oss tegne en kjapp skisse



La oss se på et nytt eksempel

```
for i in [1, 2, 3, 4]:
    print i**2
```

Her sier jeg at vi skal løkke over listen `[1, 2, 3, 4]`, og for hvert element som behandles skal vi kalle det `i`. Det vi gjør med hvert element er å skrive ut kvadratet av tallet. Resultatet blir som følger

```
1
4
9
16
```

Et annet mulig eksempel kan være å summere tallene fra 1 til 10:

```
s = 0
for i in [1,2,3,4,5,6,7,8,9,10]:
    s += i

print s
```

I det siste tilfellet løkker vi over tallene 1 til 4. Det er ofte vi har lyst til å løkke over en liste med tall, og det kan være slitsomt å skrive ut disse listene. Tenk deg for eksempel at du vil løkke over tallene fra 1 til 100 og summere dem, det er en veldig slitsom liste å skrive. Heldigivs finnes det en funksjon som gjør dette for deg, den heter `range`. Range tar to argumenter: `range(start, stop)`, og returnerer listen med tall fra og med start til (men ikke med!) stop. La oss se på noen eksempler

```
print range(1,10)
```

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

```
print range(2,8)
```

```
[2, 3, 4, 5, 6, 7]
```

Vi kan også legge med et tredje argument, som gir økningen fra hvert tall til det neste

```
print range(1,13,2)
```

```
[1, 3, 5, 7, 9, 11]
```

Vi kan nå forenkle sumeringen av tallene fra 1 til og med 10 til følgende

```
s = 0
for i in range(1,11):
    s += i

print s
```

Og vi kan summere alle heltall under 1000 som følger

```
s = 0
for i in range(1,1000):
    s += i

print s
```

Med range kan vi også lett gjenta en kodebit et gitt antall ganger. Tenk for eksempel på kortstokken vår tidligere, vi trakk 5 kort fra kortstokken med følgende kode

```
hand.append(deck.pop())
hand.append(deck.pop())
hand.append(deck.pop())
hand.append(deck.pop())
hand.append(deck.pop())
```

Denne koden kan vi også skrive som følger

```
for i in range(5):
    hand.append(deck.pop())
```

Her løkker vi over lista `[0,1,2,3,4]`, men koden vi kjører gjør ikke noe med elementene i lista, det er bare et lite triks vi bruker for å slippe å gjenta koden vår mange ganger.

La oss se på hvordan vi kan forenkle det å lage kortstokken. Sist gjorde vi det bare ved å skrive ut alle mulige kort i kortstokken

```
from random import shuffle

# Define our complete deck of cards
deck=[
's2', 'c2', 'd2', 'h2', 's3', 'c3', 'd3', 'h3', 's4', 'c4', 'd4', 'h4',
's5', 'c5', 'd5', 'h5', 's6', 'c6', 'd6', 'h6', 's7', 'c7', 'd7', 'h7',
's8', 'c8', 'd8', 'h8', 's9', 'c9', 'd9', 'h9', 'sT', 'cT', 'dT', 'hT',
'sJ', 'cJ', 'dJ', 'hJ', 'sQ', 'cQ', 'dQ', 'hQ', 'sK', 'cK', 'dK', 'hK'
'sA', 'cA', 'dA', 'hA']

# Randomize the order of the cards
shuffle(deck)
```

Siden vi skal lage kort av alle mulige valører (2 til ess) for hver av de mulige fargene (spar, kløver, ruter, hjerter), så er dette noe vi kan gjøre enklere med for-løkker. I dette tilfellet kommer vi faktisk til å bruke to for-løkker, inni hverandre

```
from random import shuffle

# Define our complete deck of cards
deck = []
for suit in ['s', 'c', 'd', 'h']:
    for value in ['2','3','4','5','6','7','8','9','T','J','Q','K','A']:
        deck.append(suit+value)

# Randomize the order of the cards
shuffle(deck)
```

Dette eksempelet er mye mer komplisert siden vi har to for-løkker, inni hverandre. Men idéen er ganske grei, den første for-løkke sier bare at vi skal ha med alle fargene, og den andre løkka sier at vi skal ha med alle valørene. Hvis du ikke helt greier å se for deg rekkefølgen kortene lages i, så er det kanskje en idé og bare skrive ut kortene etterhvert som de lages

```
for suit in ['s', 'c', 'd', 'h']:
    for value in ['2','3','4','5','6','7','8','9','T','J','Q','K','A']:
        print suit+value
```

Her får du følgende output

```
s2
s3
s4
s5
s6
s7
s8
s9
sT
sJ
sQ
sK
sA
c2
c3
c4
.
.
.
hJ
hQ
hK
hA
```

Så du ser at først lages alle de forskjellige spar kortene, så lagges alle kløver kortene osv.

Oppgaver til uke 3

Oppgave 1 — Terningkast

Bruk funksjonen `randint` til å simulere følgende terningkast

(a) En sekssidet terning

(b) En 20-sidet terning

(c) Tre sekssidede terninger

(d) Tre mynter

Oppgave 2 — Lotto

I vanlig Lotto ved Norsk tipping trekkes det syv hovedtall fra totalt 34 tall, i tilegg trekkes det 3 bonustall. Du skal lage et program som simulerer dette.

(a) Lag en liste av de forskjellige lotto-tallene.

(b) Bruk `shuffle()` og `.pop()` til å trekke ut 7 hovedtall og 3 bonustall.

(c) Skriv ut hovedtallene til skjermen, så bonustallene.

(d) Bestem deg for en lotto-rekke (det vil si syv tall) og skriv ned på papir. Kjør programmet ditt 5 ganger, hva er det beste resultatet for rekka du valgte?

Oppgave 3 — Løkker for hånd

(a) For hvert program, si hva programmet kommer til å skrive ut. Ikke kjør programmene, prøv istedet å gå igjennom koden "for hånd" for å se hva som skjer.

```
print range(2,10)
print range(1,5,3)
print range(0,10,3)
```

```
for i in range(1,5):
    print i**2
```

```
i = 0
while i < 10:
    i += 2
    print i
```

```
i = 2
while i < 10:
    i = i**2 + 1
    print i
```

(b) Kjør programmene for å sjekke om du hadde rett.

Oppgave 4 — Løkker

(a) Skriv en `while`-løkke som slår mynt og kron helt til man har slått 5 kron

(b) Skriv en `while`-løkke som slår mynt og kron 10 ganger, og skriver ut antall kron du fikk.

(c) Skriv en `for`-løkke som slår mynt og kron 10 ganger, og skriver ut antall kron du fikk.

(d) Skriv en `for`-løkke som slår mynt og kron 10 ganger, og skriver ut antall kron du fikk.

Oppgave 5 — Gjett et tall

(a) Skriv et program som trekker et tilfeldig tall mellom 1 og 100.

(b) Utvid programmet ditt så det får brukeren til å gjette på det tilfeldige tallet. Skriv en beskjed som lar brukeren vite om de gjettet riktig eller ikke.

(c) Endre programmet slik at brukeren får vite om de gjettet for høyt eller for lavt hvis de bommer.

(d) Utvid programmet ditt med en `while`-løkke, slik at brukeren får gjette igjen helt til de treffer riktig svar. For hvert gjett skal programmet si ifra om de gjettet for høyt eller for lavt.

Oppgave 6 — Summer

(a) Finn summen av tallene fra 1 til 100.

(b) Finn summen av tallene fra 1 til 1000.

(c) Skriv en funksjon som finner summen av tallene fra 1 til n.

Oppgave 7 — Krig!

Vi skal nå lage en enkel versjon av kortspillet krig. Du burde gjøre dette gradvis, og teste koden din nøye for hvert teg.

(a) Implementer en kortstokk som en liste i programmet ditt. Se i teksten om du ikke er sikker på hvordan dette gjøres

(b) Endre programmet ditt så det er to kortstokker, en som er din, og en som er motstanderen sin. Stokk begge listene med `shuffle()` fra `random`.

(c) Lag en funksjon, `slag`, som bruker `pop` til å fjerne toppkortet i begge kortstokkene og skriver dem ut til skjermen.

(d) Utvid funksjonen din, så den også finner ut hvilken av kortene som 'slår' det andre, ved å sammenligne valøren dems. Her må du bli litt kreativ med if-testen for å sjekke hvilket kort som er 'høyest'.

(e) Gjør sånn at funksjonen din legger begge kortene inn i kortstokken til vinneren. Deretter stokker du begge kortstokkene.

(f) Hvis kortene i et enkelt slag er like bra, så blir det krig. Lag en funksjon, `krig`, der du popper 3 kort fra begge kortstokkene, og sammenligner verdien til det tredje kortet fra stokkene. Gi alle kortene til vinneren.

(g) Lag en `while`-løkke som gjør at du kan fortsette å spille slag helt til du enten er tom for kort, eller du har alle kortene. Hver gang løkka-kjøres bør du bruke en `raw_input()`, slik at brukeren av programmet kan trykke Enter for å spille et nytt slag.

(h) Pynt på programmet ditt. Når programmet starter kan du skrive ut en større beskjed, og det bør være en fin beskjed etter hvert slag som forklarer hvilke kort som ble vist og resultatet av slaget. Rydd også i koden så den er fin og ryddig.

Oppgave 8 — Primtall (Utfordrende!)

Et primtall er et tall som bare er delbart med seg selv og 1. Det vil si at de ikke kan deles på andre tall uten å få desimaler. De første 10 primtallene er

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29.$$

(a) Skriv en funksjon som tar et tall n som argument, og sjekker om det er et primtall. Funksjonen skal returnere `True` hvis n er et primtall og `False` hvis det ikke er det.

(b) Bruk funksjonen din til å skrive ut alle primtall under 1000. Som kontroll på at du har gjort riktig kan du sjekke at det er 168 av dem.

Oppgave 9 — Fibonacci-tall (Utfordrende!)

Fibonnaci-tallene starter som dette

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

Måten vi finner neste tallet i rekken er at vi legger de to forrige tallene sammen. Neste tall i rekka blir altså $89 + 144 = 233$. Skrevet mer matematisk, så kan vi si at det n-te Fibonnaci-tallet er gitt ved

$$F_n = F_{n-1} + F_{n-2},$$

der $F_1 = 1$ og $F_2 = 1$.

(a) Skriv en funksjon som tar n som argument, og returnerer det n-te Fibonnaci-tallet. **(Spør gjerne om hjelp her)**

(b) Bruk funksjonen din til å skrive ut de første 50 Fibonnaci-tallene.

Videre lesing og oppgaveløsning

Hvis du har kommet deg så langt som dette, bra jobba! Jeg kommer til å utivde dette skrivet etterhvert som kurset går, så hvis du sjekker tilbake om en uke eller to er det nok en del mer info og fler oppgaver her. Men hvis du ikke klarer å vente på det, så skal jeg komme med et par kilder du kan bruke for å lære mer med en gang.

Nettkurs fra Code Academy

Først så har vi Code Academy, en nettside som gir gratis nettkurs i mange forskjellige programmeringsspråk. Code Academy er helt genialt fordi de underviser programmering hands on, som vil si at du lærer ved å kode fra første sekund. Her er en link til begynnerkurset dems in Python

`http://www.codecademy.com/tracks/python`

Programmeringsnøtter fra Project Euler

Hvis du er ute etter en utfordring, så kan jeg anbefale Project Euler. Dette er en nettside som legger ut mattenøtter med en vri, vrien er at problemene skal løses med programmering. Ta en titt på oppgavene her, de stiger raskt i vansklighetsgrad, så start på de første!

`https://projecteuler.net/problems`