Mer innføring i programmering med Python

I forrige dokument gikk vi igjennom grunnleggende Python-syntax, og lærte å lage variabler, regne i Python, og om lister, løkker og funksjoner.

Vi bygger nå videre med nye konsepter. Vi starter med if-tester, og dekker deretter plotting og while-løkker.

If-tester

Vi skal nå begynne å se på tester og logikk. Disse konseptene er viktige for at programmene våres skal kunne forgrene seg og gjøre forskjellige ting basert på forskjellig input fra brukeren eller basert på forskjellige utregninger den har gjort.

Den enkleste formen for if-test i Python skrives som følger

```
if <betingelse>:
    <kode som utføres hvis betingelsen er sann>
```

Her er `<betingelse>` noe som kan tolkes som enten *sant* eller *falskt*. Her kan man for eksempel sjekke om to ting er lik hverandre, eller sjekke om et tall er større eller mindre enn et annet tall. I Python kan vi skrive slike betingelser rett ut, dette er et eksempel på noe som kalles *bools logikk*

In [1]:

```
a = 2
b = 3

print(a < b)
```

```
True
```

In [2]:

```
if a < b:
    print("a er mindre enn b!")
```

```
a er mindre enn b!
```

Her skrives setningen kun ut hvis det faktisk er slik at a er mindre enn b. Kanskje vi ønsker å skrive ut det motsatte om det er slik at testen *ikke* er sann, da legger vi til en *else*-blokk. "If" og "else" er engelske ord og betyr "hvis"-"ellers".

In [3]:

```
a = 3
b = 2

if a < b:
    print("a er mindre enn b!")
else:
    print("a er ikke mindre enn b.")
```

```
a er ikke mindre enn b.
```

Det første eksemplet vi så på hadde kun en *if*-test. Om testen ikke er sann hopper programmet bare over den kodeblokken og går videre. Dette kan vil kalle en *hvis-så* test. Det andre eksempelet har også en *else*-blokk, og da kan vi heller si *hvis-ellers* eller eventuelt *hvis-hvis ikke*. Da er det sånn at *enten* if-blokken *eller* else-blokken kjøres. Det vil aldri skje at begge to kjører, eller at ingen av dem kjører.

Merk at vi har skrevet "a er ikke mindre enn b" i *else*-blokken, istedenfor å skrive "b er mindre enn a". Dette er fordi testen vil fortsatt være falsk om a og b er like store. Om vi vil sjekke om noe er større eller like stort som noe kan vi bruke `>=` og tilsvarende for `<=` . Dette er det samme som å skrive $\leq$ og $\geq$ matematisk.

Vi kan også lage en test som har mer enn to utfall, da bruker vi `elif` for alle de ekstra alternativene. Si for eksempel at vi har et spill med to spillere. På slutten skal vi går vinneren basert på hvem som har mest poeng.

In [4]:

```
poeng_a = 320
poeng_b = 280

if poeng_a > poeng_b:
    print("Spiller A er vinneren!")
elif poeng_a == poeng_b:
    print("Det er uavgjort!")
else:
    print("Spiller B er vinneren!")
```

```
Spiller A er vinneren!
```

Her sjekkes først den første testen, hvis den er sann kjøres koden i den blokka og resten hoppes over. Hvis den første blokka er falsk, så sjekkes den neste, og så videre. Til slutt, hvis ingen andre blokker har kjørt, så kjøres *else*-blokka. Merk at testene ikke trenger å være ekskluderende. Vi kan for eksempel først sjekke om spiller A vinner med minst 100 poeng, og så sjekke om han vant.

In [5]:

```
poeng_a = 480
poeng_b = 220

if poeng_a > poeng_b + 100:
    print("Overhvelmende seier til spiller A!")
elif poeng_a > poeng_b:
    print("Seier til spiller A!")
else:
    print("Seier til spiller B!")
```

```
Overhvelmende seier til spiller A!
```

Her er begge de to første betingelsene sanne, men det er bare den første blokka som kjøres, for så fort en blokk er kjørt, hopper vi over resten.

Testbetingelser

Her er noen vanlige testbetingelser

Matematisk symbol	Kode	Tolkning
$a < b$	a er større enn b	a > b
$a > b$	a er mindre enn b	a < b
$a = b$	a og b er mindre	a == b
$a \leq b$	a er mindre eller lik b	a <= b
$a \geq b$	a er større eller lik b	a >= b
$a \neq b$	a er ulik fra b	a != b

Merk at hvordan *større enn* og *mindre enn* tolkes avhenger av hva slags type variabler vi snakker om. For tall er det jo ganske greit, men hva med for eksempel tekst? Her vil Python bruke den alfabetiske sorteringen, slik at for to tekststrenger vil `a < b` være sant hvis `a` ville blitt sortert før `b` om vi sorterterte dem alfabetisk.

Inversjon

Når vi lager en test kan vi slenge inn en `not` for å invertere testen. For eksempel vil `if not a < b` være ekvivalent med `if b <= a`.

Boolsk logikk med `and` / `or`

Om vi ønsker å sjekke to betingelser på én gang kan vi også gjøre dette. Da legger vi til en `and` i testen, for eksempel:

```
if a < b and b < c
```

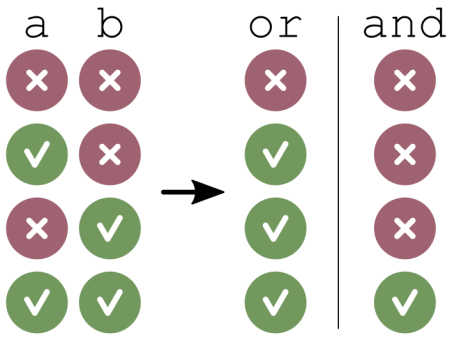
Siden vi skriver `and` så vil if-testen være sann hvis og bare hvis *begge* testene er sanne, i dette tilfellet ser vi altså at `b` må ligge imellom `a` og `c`.

Kanskje vi vil ha en test der det holder at én av dem er sanne? Dette gjør vi ved å legge til `or`

```
if a < b or b < c
```

Her betyr *or* "eller", her holder det altså at `b` er større enn `a`, eller at `b` er mindre en `c`, om en av disse er sanne så regnes testen som sann. Er begge to sanne regnes også testen som sann.

Disse betingelsene og måtene å kombinere dem på kalles gjerne Boolsk logic, etter matermatikeren George Boole. Bolsk logikk er et av fundamentene for datamaskiner, for på det laveste nivået foregår alt som 0 og 1, der 0 kan tolkes som *falskt* og 1 som *sann*. Det å behandle og manipulere disse tallene går altså stort sett ut på å bruke boolsk logikk riktig.



Vanlig feil: Å sjekke om to ting er like

Merk i tabellen at vi skriver `==` for å sjekke om verdien til to variabler er like hverandre. Dette er fordi vi bruker ett likhetstegn for å sette en variabel, og derfor blir Python forvirret om vi skriver `if poeng_a = poeng_b:`, for her tror Python vi ønsker å definere en ny variabel, og det vil vi jo ikke. Dette gir forøvring en `SyntaxError`. Det er veldig lett å skrive ett likhetstegn istedenfor to, og dette er derfor en veldig vanlig feil:

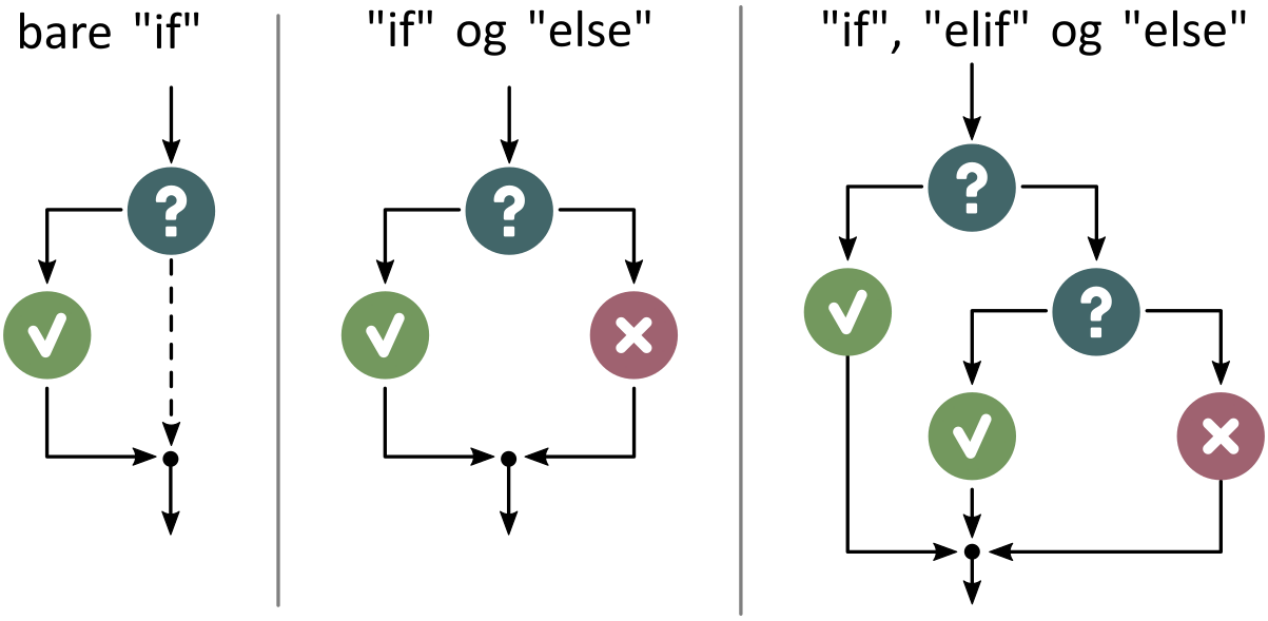
In [6]:

```
1 if poeng_a = poeng_b:
2     print("Uavgjort!")
```

```
  File "<ipython-input-6-e219412fbecb>", line 1
    if poeng_a = poeng_b:
               ^
SyntaxError: invalid syntax
```

Progamflyt i if-tester

Vi kan skissere flyten av programmet i de tre tilfellene som følger



I tilfellet helt til venstre har vi kun en `if`-blokk. Denne kjøres om betingelsen er sann, men om betingelsen er falsk så skjer ingenting, vi bare hopper blokken og fortsetter. I den midterste har vi en `if`-blokk og en `else`-blokk. Nå må en av de to blokkene skje. Til slutt har vi en situasjon med tre blokker: `if`, `elif` og `else`. Her testes først det første betingelsen, om den er sann skjer `if`-blokka og vi er ferdig. Om den første betingelsen er falsk, så sjekkes neste betingelse. Dette forklarer også hvorfor vi skriver `elif`, det er en sammenslåing av `else if`. For vi kunne også laget den samme tre-veis testen som følger

```
if <betingelse 1>:
    <kode>
else:
    if <betingelse 2>:
        <kode>
    else:
        <kode>
```

Eksempel: Quiz

La oss lage en enkel tekstbasert quiz. Programmet stiller et spørsmål til brukeren, så må brukeren skrive inn svaret sitt, så bruker vi en `if`-test til å sjekke om brukeren har rett.

In [7]:

```
svar = input("Hva er hovedtaden i Portugal?\n")
fasitsvar = "Lisboa"

if svar == fasitsvar:
    print("Riktig!")
else:
    print("Feil! Riktig svar var {}".format(fasitsvar))
```

```
Hva er hovedtaden i Portugal?
Madrid
Feil! Riktig svar var Lisboa
```

Her kan vi forbedre quizzen vår ved å legge til flere spørsmål, og så holde styr på antall poeng brukeren har. Om man lager mange spørsmål har man kanskje også lyst til å stokke om på rekkefølgen med spørsmål og bare bruke noen av dem. Man kan også legge til svaralternativer.

En ting som er litt dumt med quizzen vi lager er at svaret må matche *nøyaktig* med fasitsvaret, om man for eksempel svarer `lisboa` vil det slå ut som feil (Python tolker store og små bokstaver som forskjellige symboler. Dette kan vi fikse ved å skrive fasitsvaret vårt som `lisboa` og bruke metoden `.lower()` på svaret fra brukeren, som gjør at svarer tolkes som småbokstaver, uansett.

In [8]:

```
print("lisboa".lower())
print("Lisboa".lower())
print("LISBOA".lower())
```

```
lisboa
lisboa
lisboa
```

In [9]:

```
poeng = 0
svar = "LiSboa"
fasitsvar = "lisboa"

if svar.lower() == fasitsvar:
    poeng += 1
    print("Riktig!")
else:
    print("Feil! Riktig svar er Lisboa!")
```

```
Riktig!
```

Tilsvarende kunne vi tatt hensyn til andre små forskjeller som skrivefeil, ekstra mellomrom osv. Men dette velger vi å ikke ta hensyn til.

Eksempel: Tekstbaserte spill

Å lage spill med grafikk er litt komplisert, men man kan begynne å lage enkle, tekstbaserte spill. Historisk sett var de aller første dataspillene tekstbaserte, grunnet få muligheter til å implementere grafikk på gamle datamaskiner. Noen av de mest kjente er *Adventure*, som først kom ut i 1976 og *Zork* i 1977. I disse spillene får spilleren en tekst som beskriver situasjonen de er i, så må de gjøre valg og handlinger ved å skrive dem inn i terminalen.

Etterhvert som grafikk begynnte å komme på banen fikk disse spillene enkel grafikk som vistes på skjerm utifra det brukeren skrev, men brukeren måtte fortsatt skrive kommandoene sine på bunn av vinduet, og de fikk også ofte tekstbasert informasjon.

Dette bildet er fra Space Quest 3, som ble gitt ut i 1989.

Man kan begynne å lage enkle tekstbaserte spill selv. Disse blir gjerne litt korte, men det kan fortsatt være gøy for noen å lage små historier der brukerene kan ta egne valg.

Plotting

Vi skal nå se hvordan vi kan bruke Python til å tegne grafer. Når vi skal tegne en graf for hånd bruker vi et ruteark og lager et koordinatsystem. Når vi har koordinatsystemet regner vi oss frem til en rekke (x, y) punkter som vi tegner inn, deretter trekker vi streker igjennom punktene for å få kurven vår.

Å tegne en graf i Python fungerer helt likt, vi kaller det å *plotte*. Funksjonen `plot(x, y)` tar to lister som input, en med x-verdier, og en med y-verdier. Vi får tilgang på `plot` når vi importerer fra pylab.

Plotte en rett linje

La oss si vi ønsker å tegne linja

$$y = 3x - 5,$$

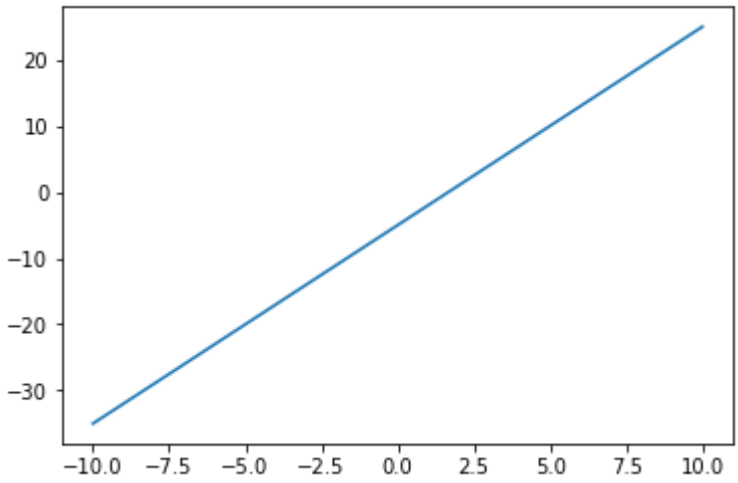
for x fra -10 til 10. Vi ser fra uttrykket at dette er en rett linje, så det holder å regne oss frem til y-verdien ved de to endepunktene, som blir

$$y(-10) = -35, \qquad y(10) = 25.$$

Da kan vi tegne kurven som følger

In [11]:

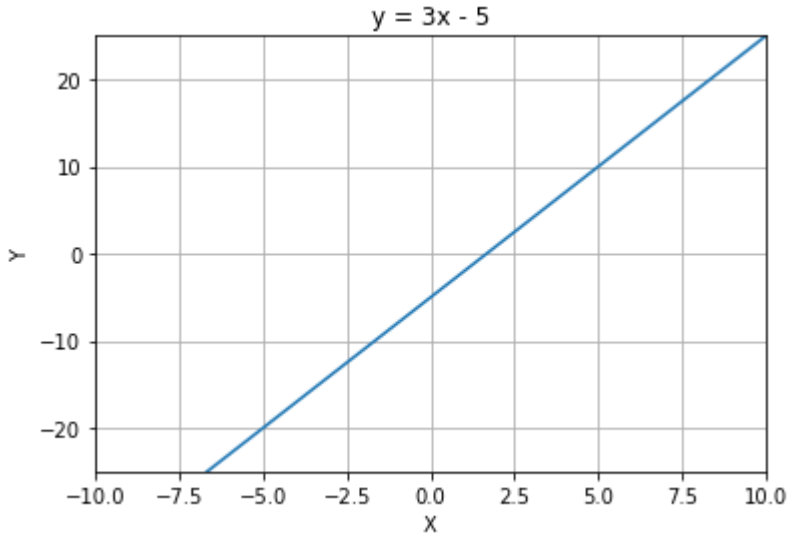
```
from pylab import *

x = [-10, 10]
y = [-35, 25]

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



Vi bruker `show` -kommandoen for å få plottet til å dukke opp på skjermen. Vi kan også pynte på plottet vårt ved å gi navn til aksene, legge på et rutenett, bestemme aksene osv. Dette må vi gjøre mellom `plot` og `show` .

In [12]:

```
plot(x, y)
title('y = 3x - 5')
xlabel('X')
ylabel('Y')
grid()
axis([-10, 10, -25, 25])
show()
```

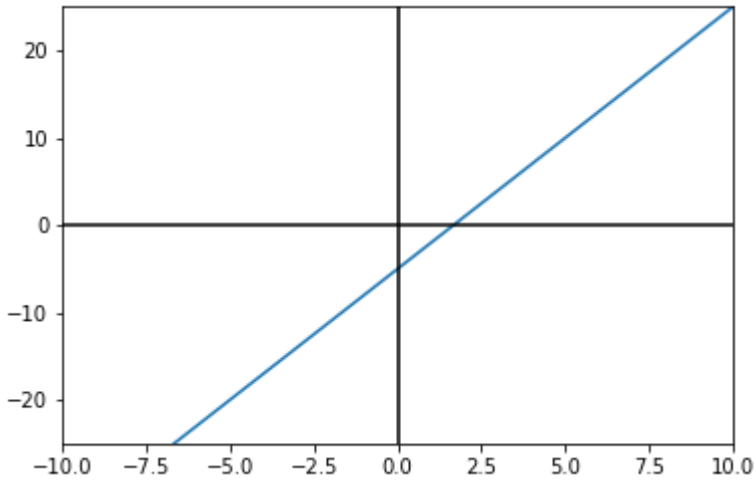


Her setter `title` , `xlabel` og `ylabel` navn på aksene og plottet, `grid()` lager et rutenett og `axis`

bestemmer vinduet ved at vi sender in en liste `[xmin, xmax, ymin, ymax]` . Kanskje vi ikke vil ha et fullt rutenett, men bare selve aksene, da kan vi legge til en vertikal og horizontal linje som følger

In [13]:

```
plot(x, y)
axvline(0, color='black')
axhline(0, color='black')
axis((-10, 10, -25, 25))
show()
```



Plotte en annengradsfunksjon

La oss prøve å plotte en funksjon som trenger litt flere punkter. Vi tar annengradsfunksjonen

$$f(x) = x^2 - 4x + 4.$$

Nå må vi regne ut en lengre rekke med x og y verdier for at vi skal få en fin kurve. Vi kunne gjort dette for hånd og plugget inn tallene i koden vår, men det tar mye arbeid. Så la oss heller gjøre det med løkker.

Først definerer vi funksjonen vår

In [14]:

```
def f(x):
    return x**2 - 4*x + 4
```

Så lager vi x og y verdiene våres ved hjelp av en for-løkke.

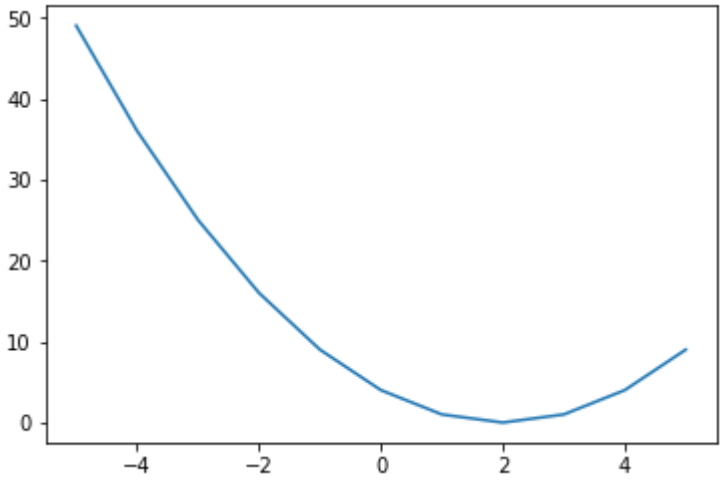
In [15]:

```
x = []
y = []

for xverdi in range(-5, 6):
    x.append(xverdi)
    y.append(f(xverdi))

print(x)
print(y)
```

```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
[49, 36, 25, 16, 9, 4, 1, 0, 1, 4, 9]
```

In [16]:

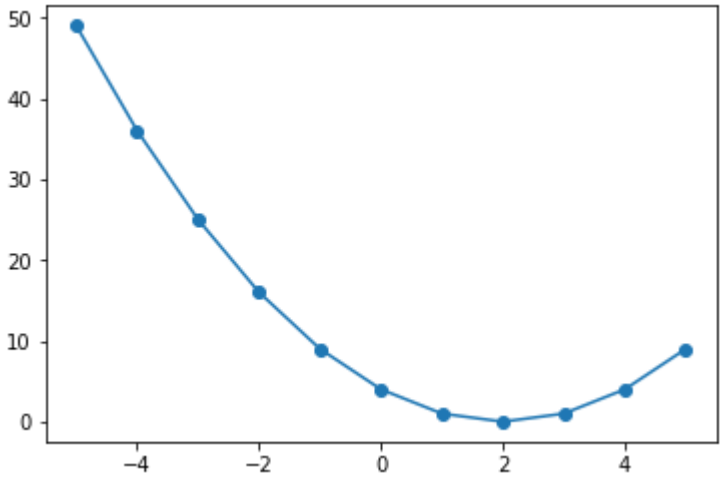
```
plot(x, y)
show()
```



Vi ser at funksjonen blir litt hakkete, dette er fordi Python trekker en rett strek mellom punktene vi gir inn. Dette kan vi vise ved å skrive `plot(x, y, 'o-')`. Her sier vi at vi skal plotte prikker på selve punktene, og en strek mellom dem: `o-`. Vi kunne også bare plottet prikker (`o`), plottet prikker med stiplede linjer mellom `o--`, bare plottet stiplede linjer `--`. Vi kunne også brukt andre former, `x` er for eksempel kryss.

In [17]:

```
plot(x, y, 'o-')
show()
```



Eksempel: Kvadrat og kubikktall

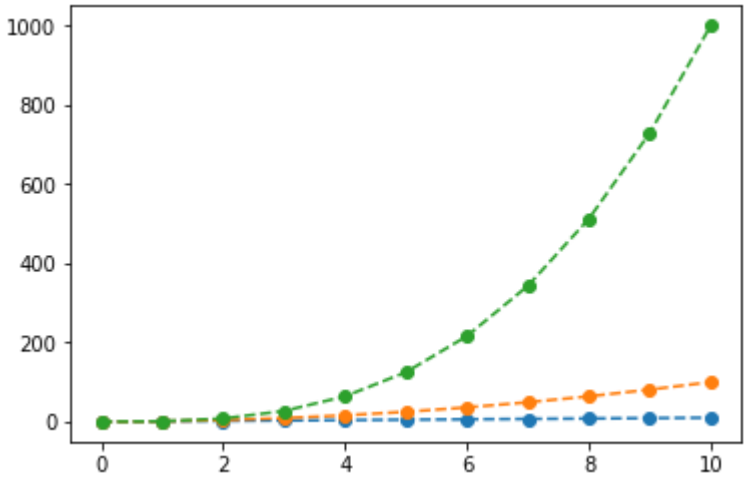
Vi vil nå se litt på hvor fort kvadrattall og kubiktall vokser. Vi regner derfor først ut de første 11 kvadrattallene og kubikktallene (medregnet 0) og plotter sammen med den rette linjen $y = x$. For å plotte flere kurver i samme figur kaller vi bare på `plot` flere ganger før vi bruker `show` . Hver kurve vil få en annerledes farge for lettere å skille på dem (vi kan også velge farge selv ved å bruke for eksempel `plot(x, y, color="blue")` .

In [18]:

```
x = []
kvadrattall = []
kubikktall = []
for i in range(11):
    x.append(i)
    kvadrattall.append(i**2)
    kubikktall.append(i**3)
```

In [19]:

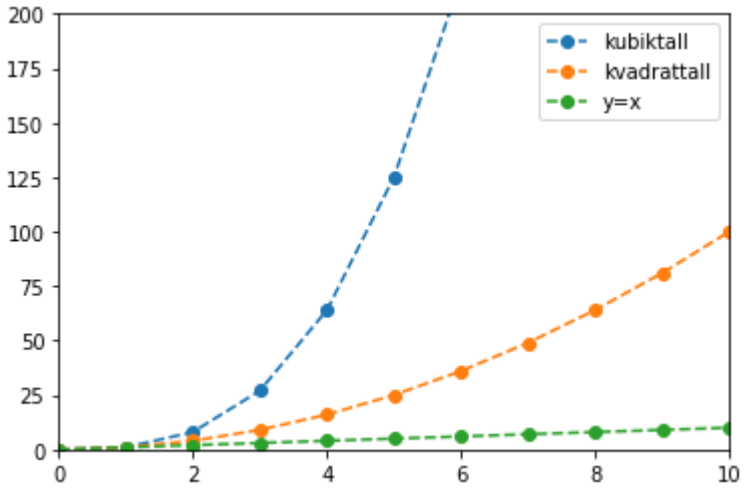
```
plot(x, x, 'o--')
plot(x, kvadrattall, 'o--')
plot(x, kubikktall, 'o--')

show()
```



Her er det ikke så vanskelig å skjønne hvilken kurve som er hvilken, men om det skulle vært det kan det være lurt å legge på en *legend*. Vi zoomer også litt inn på y-aksen ved hjelp av `axis`, så vi kan se forskjellen på de to mindre kurvene lettere.

In [20]:

```
plot(x, kubikktall, 'o--', label='kubiktall')
plot(x, kvadrattall, 'o--', label='kvadrattall')
plot(x, x, 'o--', label='y=x')
legend()
axis((0, 10, 0, 200))
show()
```



Så regner vi ut de tilhørende y-verdiene

Å plotte jevne kurver

For å få en fin og jevn kurve burde vi ha flest mulig punkter. La oss for eksempel si vi skal lage en sinuskurve (ikke ungdomsskolepensum, men et bra eksempel). La oss si vi ønsker å tegne sinus fra 0 til 360 grader. Om vi velger et punkt hver 90 grad blir ikke dette så bra.

In [21]:

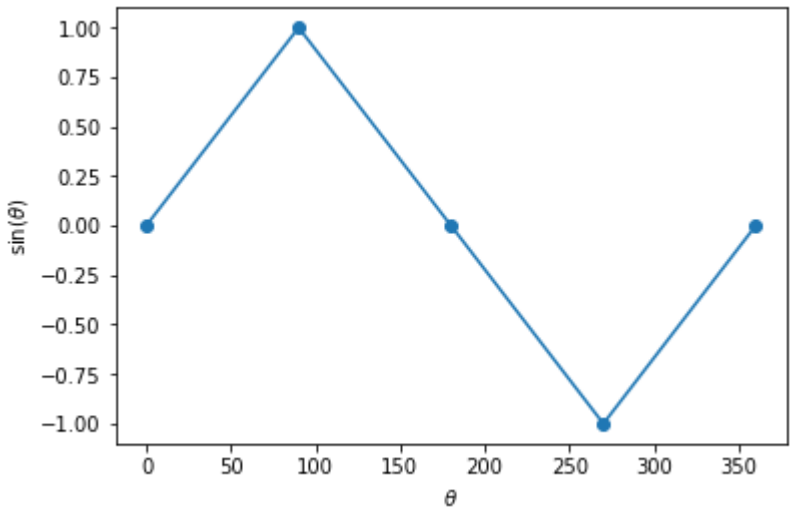
```
x = []
y = []

for vinkel in range(0, 360+1, 90):
    x.append(vinkel)
    y.append(sin(deg2rad(vinkel)))

print(x)
print(y)
```

```
[0, 90, 180, 270, 360]
[0.0, 1.0, 1.2246467991473532e-16, -1.0, -2.4492935982947064e-16]
```

Her bruker vi `deg2rad` fordi de trigonometriske funksjonene som `sin` tar radianer inn, ikke grader.

In [22]:

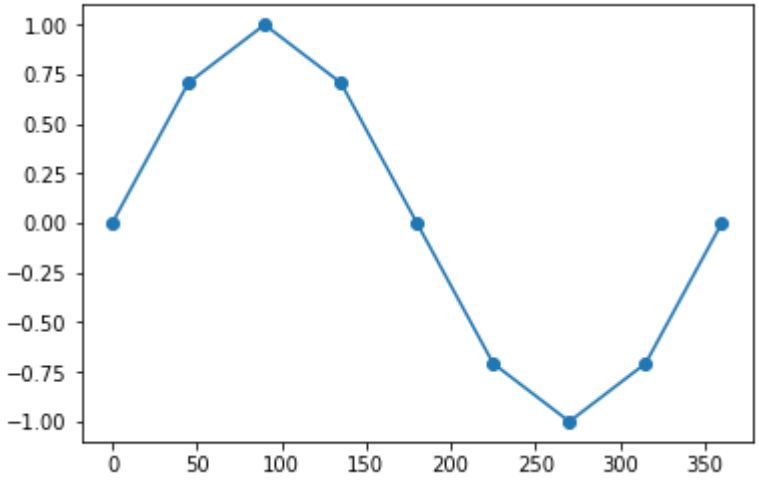
```
plot(x, y, 'o-')
xlabel(r'$\theta$')
ylabel(r'$\sin(\theta)$')
show()
```



La oss prøve hver 45 grad isteden

In [23]:

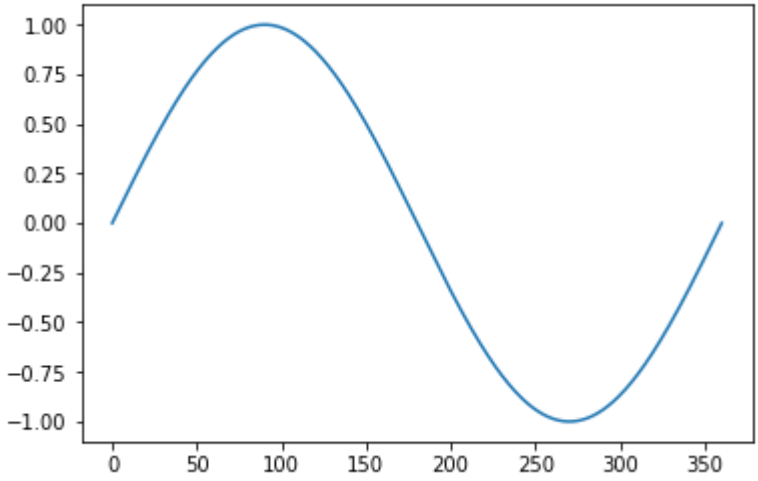
```
x = []
y = []

for vinkel in range(0, 360+1, 45):
    x.append(vinkel)
    y.append(sin(deg2rad(vinkel)))

plot(x, y, 'o-')
show()
```



Nå begynner sinuskurva å komme frem. Den er fortsatt hakkete og ikke så veldig pen, men vi ser ihvertfall mer av formen. Vi prøver nå med et punkt per grad, da vil punktene ligge så nær hverandre at de overlapper i plottet, så vi endrer samtidig til å bare plotte linjer mellom punktene.

In [24]:

```
x = []
y = []

for vinkel in range(0, 361, 1):
    x.append(vinkel)
    y.append(sin(deg2rad(vinkel)))

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



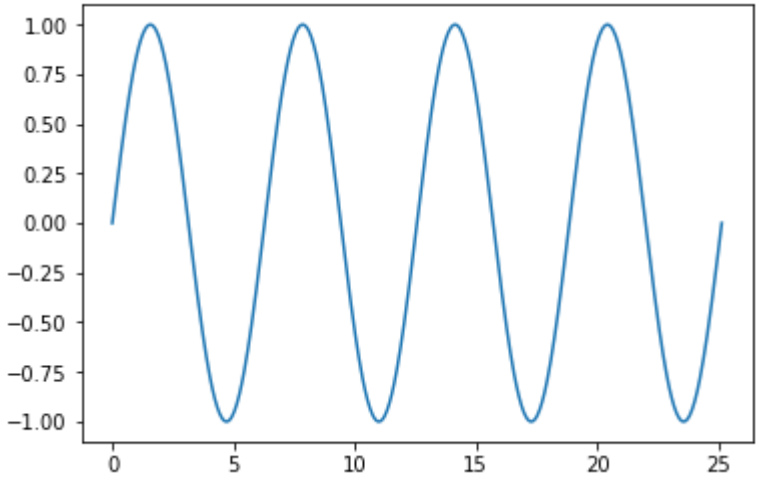
I dett eksempelet kan det godt være vi hadde fått en like fin kurve om vi hadde brukt et punkt hver 5 grad, eller tilogmed kanskje hver 10 grad, men samtidig regner jo datamaskinen veldig raskt - så vi kan like greit bruke mange punkter.

Si vi ønsket å tegne en sinuskurve i radianer isteden. La oss si vi ønsker å plotte mellom 0 og 8π, her blir jo 8π et desimaltall, og om vi ønsker hundrevis av punkter så blir steget mellom to punkter også veldig lite. Her fungerer ikke `range` lenger, fordi den funksjonen er laget for kun å fungere med heltall. Det finnes to andre nyttige funksjoner vi kan bruke i dette tilfellet.

Den første funksjonen er `arange`, som oppfører seg helt likt som range, men som tar desimaltall, her kunne vi for eksempel skrevet `arange(0, 8*pi, 0.1)`. Den andre er `linspace`, som står for *linear space*. Funksjonen tar tre verdier inn `linspace(min, max, antall)` og gir tilbake punkter som ligger jevnt fordelt mellom minimum og maksimumsverdien. Så når vi har lyst å plotte mellom 0 og $8 * pi$ kan vi for eksempel skrive `linspace(0, 8*pi, 500)` og vi får 500 punkter og en jevn kurve.

In [25]:

```
x = []
y = []

for radianer in linspace(0, 8*pi, 500):
    x.append(radianer)
    y.append(sin(radianer))

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



Man kan også bruke `arange` og `linspace` utenfor løkker.

In [26]:

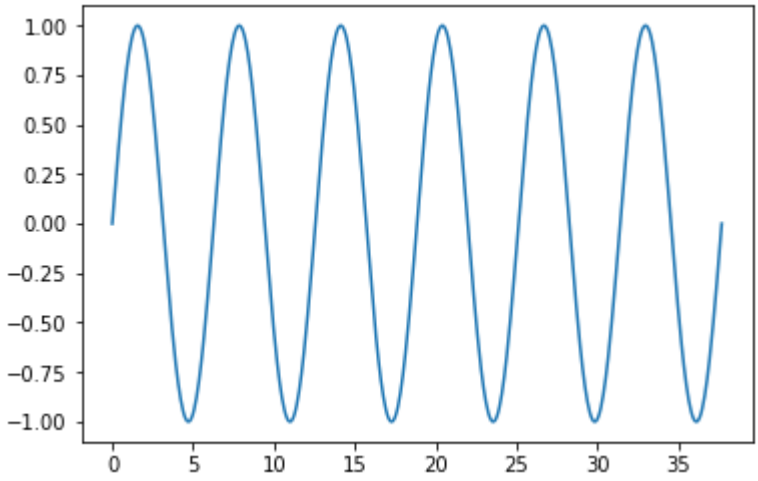
```
x = linspace(0, 10, 11)
print(x)
print(type(x))
```

```
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
<class 'numpy.ndarray'>
```

Her er en liten teknisk detalj: Det `linspace` gir her er `arrays`, dette er en datatype som er laget for å oppbevare matematiske matriser, og det er spesiallagd for å regne på ting. En fordel med dette er at ting er *vektoriserte*, som betyr at vi kan sende inn et helt array med tall til for eksempel `sin`, som gjør at man kan komme unna med mindre kode og ting går raskere:

In [27]:

```
x = linspace(0, 12*pi, 500)
plot(x, sin(x))
show()
```

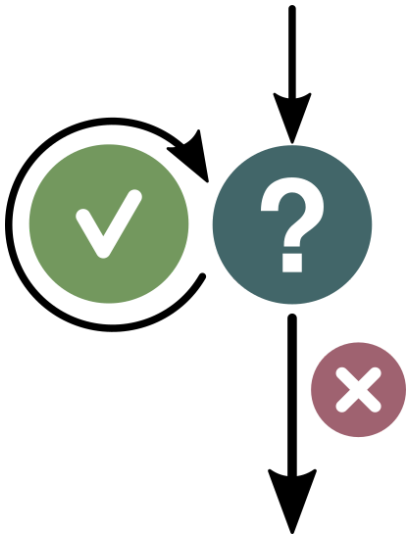


Dette kan være fint når man vet hva man driver med, men det er en mindre intuitiv og gjennomsiktig prosess for nybegynner, og spesielt med elever kan det være lurt å ta ting steg for steg ved hjelp av en løkke.

While-løkker

Vi har allerede sett på løkker i Python, nemlig for-løkker. Nå skal vi gå igjennom den andre hovedtypen med løkker som vi finner i Python: *while*-løkker. While løkker minner litt om en *if*-test, når koden kommer til while-løkka vil den sjekke en betingelse, og koden forgrener seg avhengig om betingelsen er sann eller falsk. Forskjellen fra en if-test er at etter kodeblokken er kjørt, så hopper koden tilbake av starten av løkka og sjekker betingelsen på nytt. Om betingelsen fortsatt er sann, så kjøres løkka på nytt. Så sjekkes løkka på nytt osv.

Vi kan tegne situasjonen som dette:



Syntaksen for en *while*-løkke er som følger

```
while <betingelse>:
    <kodeblokk>
    <kodeblokk>
    <kodeblokk>
```

Løkken har altså akkurat samme form som en if-test uten en else-blokk (eller "hvis-så", som vi kalte den), forskjellen er at kodeblokken kan gjentas mange ganger.

La oss se på et eksempel:

In [28]:

```
teller = 0

while teller < 10:
    teller += 1
    print(teller)
```

```
1
2
3
4
5
6
7
8
9
10
```

Her lager vi først en tellevariabel, så har vi en while-løkke der betingelsen er at `teller < 10`. Siden vi definerte `teller` til å starte som 0 vil denne testen være sann. Da kjøres kodeblokken, her økes `teller` til 1 og vi skriver den ut, så vi får 1 på skjermen. Så sjekkes betingelsen på nytt, `1 < 10` er også sant, så vi øker telleren til 2 og skriver det ut til skjermen. Slik fortsetter det helt til betingelsen sjekker at `9 < 10` som sann, så vi legger til 1, får 10, skriver dette ut, og så sjekker vi og får `10 < 10` som gir falskt. Dermed blir 10 det siste som skrives ut, og ikke 9 eller 11.

Vanlig feil - Uendlige løkker

Når man lager en for-løkke er det alltid et *endelig* antall repetisjoner, for vi løkker over en liste med en bestemt løkke. Med en *while*-løkke derimot, har vi ingen garanti for at løkken ikke vil gjenta seg selv uendlig mange ganger. Ta eksempel over for eksempel, la oss si vi glemmer linjen `teller += 1` og bare skriver

```
teller = 0
while teller < 10:
    print(teller)
```

Hva skjer nå? Jo, telleren skrives ut, som er `0`, så sjekkes betingelsen på nytt `0 < 10` er fortsatt sann, så vi gjentar løkka. Sånn fortsetter det inn i uendeligheten. Det gjelder altså å være litt forsiktig med uendelige løkker.

Uansett hvor forsiktige vi er, så kommer vi til å lage en uendelig løkke ved uhell en gang. Når det skjer, så må vi gå inn og avbryte kjøringen manuelt, i Jupyter kan vi gjøre dette ved å klikke på `Stopp` -symbolet i verktøylinja, eller klikke oss inn på `Kernel` -> `Interrupt`. Etter man har avbrutt kjøringen kan man fikse på løkka og prøve igjen.

Uendlige løkker med vilje

Noen ganger vi man faktisk lage uendelig løkker med vilje. Dette kan man gjøre ved å skrive

```
while True:
    <kodeblokk>
```

Her vil "betingelsen" sjekkes hver gang og være sann, slik at koden bare gjentar og gjentar seg selv. Vi kan inne i koden avbryte en slik løkke med `break` som bryter ut av en løkke (også for-løkker). Ta for eksempel dette programmet som ruller en terning hver gang brukeren trykker enter, men som avslutter om man skriver `slutt`.

In [29]:

```
while True:
    svar = input("Trykk enter for å rulle en terning. Skriv `slutt` for å avslu
    if svar == "slutt":
        break
    else:
        print(randint(1, 6)+1)
```

```
Trykk enter for å rulle en terning. Skriv `slutt` for å avslutte.
2
Trykk enter for å rulle en terning. Skriv `slutt` for å avslutte.
2
Trykk enter for å rulle en terning. Skriv `slutt` for å avslutte.
3
Trykk enter for å rulle en terning. Skriv `slutt` for å avslutte.
5
Trykk enter for å rulle en terning. Skriv `slutt` for å avslutte. slut
t
```

For- eller while-løkke?

Vi har nå sett begge typene med løkker i Python. De to er noe forskjellig, men ikke voldsomt. Det er faktisk slik at alle typer problemer man kan løse med en for-løkke kan man også gjøre med en while-løkke, og motsatt. Det er derimot ikke sikkert at koden blir like intuitiv eller elegant med begge variantene.

For-løkker er nyttige når vi vet akkurat hva vi ønsker å løkke over, for eksempel en bestemt tallrekke, eller vi vet hvor mange ganger noe skal gjentas. While-løkker er mer nyttig når vi *ikke* vet hvor mange ganger noe skal gjentas, vi bare vet hva vi ønsker å oppnå.

Det er også debatt om hvilke løkker det er lurest å lære bort først. Noen mener at *while*-løkker er mer intuitive og lette og forstå seg på, mens andre mener *for*-løkker er mer rett frem. Her er det ingen fasitsvar. Det som derimot *er* lurt, er å dekke if-testen før while-løkker og lister før for-løkker.

Eksempel: Høyrentekonto

Når vi diskuterte for-løkker så vi på et eksempel der vi regnet ut hvor mye penger som var på en sparekonto for hvert år som gikk, når vi begynnte med 10000. Nå kan vi snu spørsmålet på hodet og si: *hvor mange år må vi vente før vi har 20000 kroner på kontoen?*. While-løkker er perfekte for å besvare spørsmål av denne typen.

Vi kan starte med å definere en variabel `penger` som vi setter til våre opprinnelig 10000 kroner. Vi kan nå

lage en løkke med betingelsen `while penger < 20000` . Inne i løkka må vi legge på den årlige renta og oppdatere `penger` , på denne måten vil mengden penger øke for hver *iterasjon*, og vi unngår en uendelig løkke. I tillegg må vi ha en variabel som teller antall ganger løkka har gjentatt seg, for det er jo dette vi egentlig er ute etter.

In [30]:

```
penger = 10000
år = 0

while penger < 20000:
    penger *= 1.035
    år += 1

print("Du må vente i {} år, og det er da {:.0f} kroner på konto".format(år, pen
```

```
Du må vente i 21 år, og det er da 20594 kroner på konto
```

Her har vi valgt å bare skrive ut sluttsvaret, men man kan også fint legge inn en `print` inne i løkka og se hvordan pengemengden vokser etterhvert.

Plotte høyrentekonto

La oss gå inn og plotte pengene på konto for hvert år i forrige eksempel. Når vi skal plotte alle årene istedet for å skrive ut må vi ha lister istedenfor tall. Her finnes det to måter vi kan gjøre dette på. Vi kan enten ha listevariabler *i tilegg* til de vi hadde originalt:

In [31]:

```
penger = 10000
år = 0

penger_liste = []
år_liste = []

while penger < 20000:
    penger *= 1.035
    år += 1

    penger_liste.append(penger)
    år_liste.append(år)
```

Eller vi kan kun bruke lister, og bruke `[-1]` for å hente ut resultatet fra forrige iterasjon

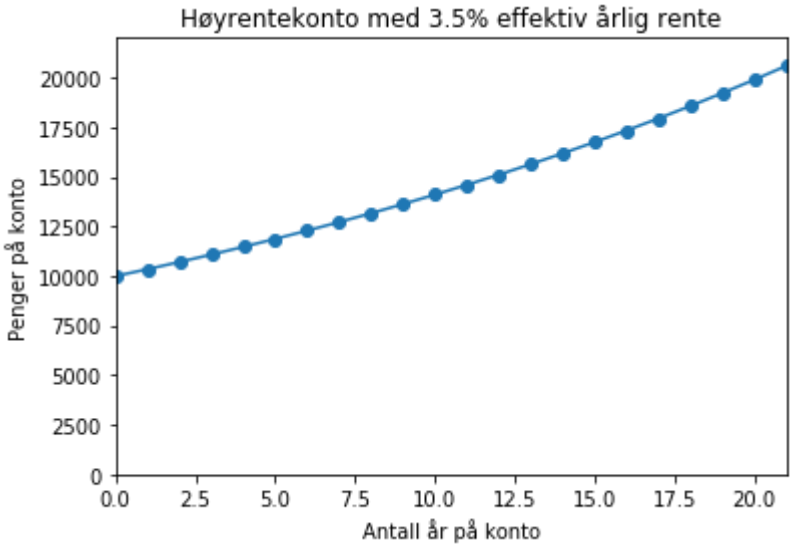
In [32]:

```
penger = [10000]
år = [0]

while penger[-1] < 20000:
    penger.append(1.035*penger[-1])
    år.append(år[-1] + 1)
```

Uansett hvilken måte vi foretrekker blir resultatene like

In [33]:

```
plot(år, penger, 'o-')
xlabel('Antall år på konto')
ylabel('Penger på konto')
axis((0, år[-1], 0, 22000))
title('Høyrentekonto med 3.5% effektiv årlig rente')
show()
```



Eksempeloppgave: While-løkker for hånd

Gå igjennom løkkene under for hånd og forutsi hva de kommer til å skrive ut. Etterpå kan du kjøre dem og se om du hadde rett.

Oppgave a)

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

Oppgave b)

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

Oppgave c)

```
a = 0
b = 0
c = 0

while c < 10:
    a += 1
    b += a
    c += a + b

print(a)
print(b)
print(c)
```