Programmering med Python

Per G. Østerlie
FLT, NTNU
per.g.osterlie@ntnu.no

2. februar 2024

Innhold

Siden denne teksten er litt kjapt satt sammen kan det forekomme noen småfeil (sikkert store også). Gi meg beskjed i så fall, så kan de rettes opp.

1 Introduksjon

1.1 Et eksempel på et Python-program

Vi kan starte med å se på et ferdig program som er skrevet i programmeringsspråket Python.

```
1  print("Programmet finner løsning på andregradslikninger på formen ax^2 + bx
       + c = 0")
2  a =  float( input("a: "))
3  b =  float( input("b: "))
4  c =  float( input("c: "))
5
6  d = b**2-(4*a*c)
7
8  if d > 0:
9      x_1 = (-b + d**0.5)/(2*a)
10     x_2 = (-b - d**0.5)/(2*a)
11     print("Løsningene er ", x_1, " og ", x_2)
12 elif d == 0:
13     x_1 = (-b/2*a)
14     print("Løsningen er ", x_1)
15 else:
16     print("Ingen reelle løsninger")
```

Programkode 1.1: abc-formelen

Ved første møte er slik kode uforståelig for oss. Vi må lære oss språket – akkurat som om vi skal snakke med personer med andre språk enn norsk. Språk er bygd opp av semantikk, grammatikk, ord og syntaks. Dette er noe vi må lære. Siden vi kan engelsk fra før, ser vi at noen av ordene er kjente: `input`, `print` og `if`. Utviklerne har valgt disse ordene nettopp av den grunn. Vi kan gjenkjenne hva kommandoene gjør.

Vi kan også se at det blir utført en del matematiske operasjoner (kall det gjerne utregninger). Her står det `d = b**2-(4*a*c)`. Python krever at vi holder oss på linja og ikke benytter indekser eller brøkstrek. Det krever en skrivemåte som skiller seg fra den vi benytter i matematikk. Oversatt får vi

$$\texttt{d = b**2-(4*a*c)} \quad \longrightarrow \quad d = b^2 - 4 \cdot a \cdot c$$
$$\texttt{x\_1 = (-b + d**0.5)/(2*a)} \quad \longrightarrow \quad x_1 = \frac{-b+\sqrt{d}}{2 \cdot a}$$

Det betyr at vi også må lære oss en nesten helt ny notasjon for å kunne skrive kode.

I tillegg kan vi observere at noe av teksten er rykket inn. Slike innrykk er vesentlige i Python. De danner blokker for hva som skal utføres. Denne måten å markere blokker på kalles *signifikante innrykk* (eng. significant white space).

Foreløpig er dette et eksempel på hva vi skal bli kjent med. Koden er skrevet av noen som allerede kjenner kommandoene og syntaksen til Python. Før vi kommer så langt er det en del vi må se på, men til slutt vil nok det som står der være ganske greit. La oss starte litt enklere og fortsette ut fra det.

1.2 Vi lager noen enkle program

Vi starter med å lage et enkelt program i Python, og følger en god tradisjon med et program som skriver: Hallo verden. Her er det

```
"""
Dette er et enkelt program.
Mitt første forsøk på å programmere
"""

print("Hallo verden") # Skriver ut
```

Programkode 1.2: Hallo verden

Dette enkle eksemplet viser hvordan vi kan benytte programmering til å få datamaskina til å gjøre som vi vil. Med kommandoen `print("Hallo␣verden")` ber vi om å få skrevet ut teksten *Hallo verden* til skjermen. Allerede her møter vi en kommando vi vil benytte ofte: `print()`. Kommandoen vil skrive ut det vi setter inn mellom parentesene og gir denne utskrifta:

```
Hallo verden
```

Noe annet vi også støter på er datatypen *tekststreng*, som er markert med anførseltegn[1]: `"Hallo␣verden"`. Resultatet blir at denne tekststrengen skrives ut.

Legg også merke til det står noen kommentarer i starten. Mellom de to linjene med tre anførseltegn kan vi skrive hva vi vil. Ofte er det lurt å ta med noen kommentarer til seinere bruk eller slik at andre skal forstå hva koden inneholder.

Vi kan også skriver kommentarer med bare ei linje, eller etter en kommando, ved å bruke tegnet #. Når det benyttes vil ikke Python bry seg med hva som kommer etterpå. Tegnet kalles en *hashtag* eller, på norsk, en *skigard*.

En tekstvariabel La oss forandre litt på koden vår og skrive

```
tekst = "Hallo verden" # Lager en variabel av typen tekststreng

print(tekst)
```

Programkode 1.3: Hallo verden 2.0

Koden gjør akkurat det samme, men bruken av en variabel kommer tydeligere fram. Kommandoen `tekst = "Hallo␣verden"` tilordner verdien `"Hallo␣verden"` til variabelen `tekst`. I den siste linja ber om at innholdet i variabelen skrives ut.

Disse små eksemplene gjør at vi møter mye av det vi vil arbeide videre med i programmeringa: variabler og kommandoer. Vi vil se nærmere på det seinere.

Oppgave 1

Lag et program som skriver ut navnet ditt.

[1] Kalles også gåseøyne, hermetegn og likende

Et lite eksempel til Vi kan se på et annet eksempel hvor vi finner arealet av et rektangel.

```
1 bredde = 7.3
2 hoyde = 2.4
3 tekst = "Arealet er: "
4 print(tekst, bredde * hoyde)
```

Programkode 1.4: Litt mer avansert

```
Arealet er: 17.52
```

Oppgave 2

Lag et program som skriver ut omkretsen av et rektangel med gitt høyde og bredde.

2 Aritmetikk

Nå skal vi se på

- ❏ aritmetikk i Python
- ❏ operatorer

2.1 Enkle utregninger

For å eksperimenter med noen utregninger er det enklest å bare benytte det som kalles terminalen (andre navn er konsollen eller consol). Terminalen er der vi får resultatet av programmene våre. Vi kan også skrive kommandoer der å få dem utført. Fordelen med det er at vi får svaret med en gang. Alternativet er å lage program for det samme og kjøre programmet. Da må vi si fra at vi ønsker å få skrevet ut resultatet. Her er de to alternativene for å vise utregning av `1+2`

```
>>> 1+2
3
```

```
1 print(1+2)
```

For å få gjort noen kjappe utregninger er det enkleste å bare prøve ut kommandoene i terminalen, men en oppnår akkurat det samme hvis utregningene skrives inne i en `print`-kommando. I eksemplene her vil jeg benytte terminalen.

De fire regningsartene

Her er noen eksempler på utregning i Python

```
>>> 17+24 #Addisjon
41
>>> 31-12 #Subtraksjon
19
>>> 54*23 #Multiplikasjon
1242
>>> 52/7 #Divisjon
7.428571428571429
```

Legg merke til at multiplikasjonstegnet er en asterisk `*`. Ellers er skrivemåten omtrent som vi er vant med.

Oppgave 3

Prøv med forskjellige enkle utregninger med de fire regningartene.

2.2 Litt mer avansert

I Python finner vi flere aritmetiske operatorer. Tabellen viser en oversikt over de mest vanlige.

De fire regningsartene har vi sett på, men de andre må forklares.

Operatorer		
Tegn	**Operasjon**	**Eksempel**
$+$	Addisjon	`a+b`
$-$	Subtraksjon	`a-b`
$/$	Divisjon	`a/b`
$*$	Multiplikasjon	`a*b`
$**$	Potens	`x**3`
$//$	Heltallsdivisjon	`a//b`
$\%$	Modulus	`a%b`

Tabell 2.1: Operatorer

Potenser skrives ved å benytte operatoren `**`. I vanlig matematisk notasjon skriver vi 3^2. I Python må vi skrive det samme som `3**2`. Prøver vi det ser vi at svaret stemmer.

```
>>> 3**2
9
```

Hva om vi ønsker å finne kvadratrota av 2? Svaret på det er at Python ikke har innebygd en kommando for å finne røtter. Vi kan hente inn flere kommandoer, blant dem en kommando for å finne kvadratrota, men vi kan også benytte det vi vet om potenser, nemlig at $\sqrt{2} = 2^{\frac{1}{2}}$ og skrive `2**(0.5)`.

Vanlig divisjon blir utført på denne måten

```
>>> 52/7 #Divisjon
7.428571428571429
>>> 3/7
0.42857142857142855
```

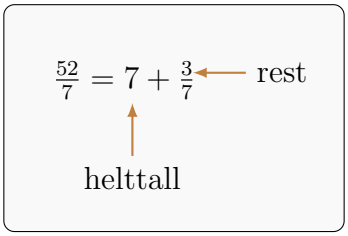
Heltallsdivisjon finner heltallet av divisjon og operatoren er //

```
>>> 52//7
7
```

Modulo , eller restdivisjon, gir oss resten som blir igjen etter en heltallsdivisjon.

```
>>> 52%7 #Modulo
3
```

Både heltallsdivisjon og restdivisjon er operasjoner som benyttes en del i programmering. Figuren under viser hvor resultatene av operasjonene kommer fra.



Oppgave 4

Er 354045614 delelig med 13?
Tips: Vil resten bli lik null?

2.3 Regnerekkefølge

Når vi skal skrive utregninger må vi gjøre det på en måte som gir mening for den som skal lese det. I programmering kommuniserer vi til ei datamaskin, som ikke kan legge til eller tolke det som står der. Parenteser for å markere regnerekkefølgen blir da svært viktig. Her er det bare å prøve å få utregningene så tydelige som mulig – og husk at det er bedre med for mange enn for få parenteser.

Oppgave 5

Regn ut i Python og kontroller svaret

a. $\frac{6 \cdot 2}{2}$
b. $\frac{6+2}{3}$

Oppgave 6

Regn ut med Python

$$3 + 0,75 + 2^3 - \sqrt{14} + \frac{3^4 - 4}{2^3 - 1}$$

Svaret skal bli: 18.998342613226058

Oppgave 7

Hva blir resultatet av denne utregninga?

```
5 + (4 - 2)* 2 + 4 % 2 - 4 // 3 - (5 - 3)/ 7
```

3 Variabler og datatyper

Nå skal vi se på

- ❏ hva en variabel er
- ❏ forskjellige datatyper
- ❏ hvordan vi kan hente inn data fra en bruker
- ❏ konvertering mellom datatyper

Programmering innebærer å ta vare på, og behandle, data. Det er nettopp derfor vi har navnet datamaskin og databehandling. Vi skal nå se på at data er og at de kan være forskjellige typer. Vi skal også se på hvordan vi kan lagre data i Python og hvordan vi kan konvertere fra den ene typen til den andre.

3.1 Programmeringsvariabler

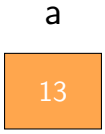
Variabler er kanskje kjent fra matematikk og andre fag. I programmering møter vi også begrepet variabler. Navnet er det samme og det er likheter med hvordan de benyttes i andre fag, men være oppmerksom på at det ikke alltid er tilfelle. I programmeringsspråk er variabler *plassholdere* ved at de tar vare på verdier. Det er derfor programmeringsvariabler ofte kalles *containers* i engelskspråklig litteratur. På norsk er *plassholdere* ofte brukt. Plassholder er egentlig å foretrekke, men jeg vil bruke «variabel», eller «programmeringsvariabel», her siden begrepet ser ut til å gå igjen i lærebøker og læreplan. Vær bare klar over at ordet ikke er entydig og betydningen vil være, nettopp, variabel.

I informatikkfaget kan vi tenke på programmeringsvariabler som en lagringsplass. La oss se på et eksempel:

```
a = 13
```

Sjøl om det er en enkel kommando skjer det mye. Python vil opprette en variabel med navnet `a`. Likhetstegnet er her en operator, som gjør at verdien som følger legges inn i variabelen. Likhetstegnet kan leses som «settes lik» og har her en rolle som en tilordning. I mange programmeringsspråk unngås flertydigheten til likhetstegnet ved å benytte := som operator, men ikke i Python.

Ofte kan det være greit å forestille seg variabelen som en boks, eller ei skuffe, hvor noe plasseres. I dette tilfellet plasserer vi verdien 13 i en boks som heter `a`. Vi kan representere det ved denne figuren:



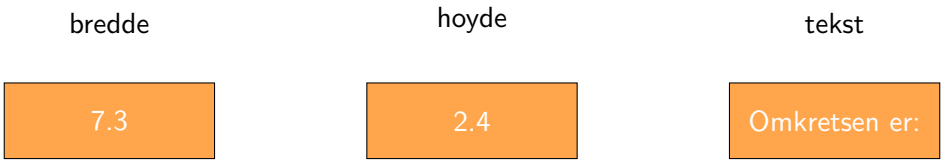
For å få ut innholdet i variabelen til skjermen benytter vi kommandoen `print()`. Skriver vi `print(a)` vil resultatet nå bli `13`.

Tidligere kom vi fram til denne koden for å finne omkretsen av et rektangel

```
1 bredde = 7.3
2 hoyde = 2.4
3 tekst = "Omkretsen er: "
4 print(tekst, 2*bredde + 2* hoyde)
```

Programkode 3.1: Omkretsen av et rektangel

Her har vi tre tilordninger i starten og vi kan tenke oss data som tilordnes legges i bokser.



I den siste kommandoen `print(tekst, 2*bredde + 2* hoyde)` hentes innholdet i boksene og benyttes i operasjonene. Til slutt skrives svaret ut.

Hva er spesielt med programmeringsvariabler?

Ved at variablene er plassholdere og at kommandoer utføres sekvensielt – steg for steg fra den første linja, kan vi skrive kode som den i programmeringskode 3.2.

```
1 a = 7
2 a = a + 3
3 print(a)
```

Programkode 3.2: Bruk av programmeringsvariabel

Noe slikt ville vært uhørt i matematikken, men siden vi her har programmeringsvariabler går det greit. Først tilordnes heltallet sju til `a`. Neste linje kan leses som: «legg til tre til det som allerede er lagret i a». Det nye innholdet i `a` blir det gamle innholdet pluss tre og programmet vil skrive ut 10. En slik bruk er svært vanlig i programmering. Ofte vil vi ha en teller som økes med en verdi, som oftest 1, og en slik bruk av programmeringsvariabler gjør det mulig:

```
1 teller = 1
2 teller = teller + 1 #Innholdet i teller økes med 1
```

Programkode 3.3: Bruk av teller

Oppgave 8

Vi har dette programmet

```
1 a = 7
2 b = 8
3 a = b
4 c = a + 1
5 print(a)
6 print(b)
7 print(c)
```

Hva blir resultatet?
Tips: Det kan være lurt å finne fram papir og blyant for å tegne bokser.

3.2 Grunnleggende datatyper

Da har vi sett litt på programmeringsvariabler. I eksemplene var de fleste variablene heltall, men det dukket opp noen desimaltall og tekststrenger også. Nå skal vi se mer på de forskjellige

datatypene og starter med de mest grunnleggende.

Noen datatyper i Python	
Forkortelse	**Type**
`int`	Heltall
`float`	Desimaltall/Flyttall
`str`	Streng
`bool`	Boolsk

Tabell 3.1: Noen datatyper i Python

Disse datatypene må vi se nærmere på. Vi starter med de som er tallverdier

Tall

Fra matematikken kjenner vi til flere typer tall som heltall og desimaltall. Et programmeringsspråk må kunne lagre de forskjellige typene og behandle dem ut fra de egenskapene tallene har.

Heltall

På engelsk kalles heltallene for *integer* og det er grunnen til at navnet Python benytter er forkortelsen `int`. I matematikk benyttes symbolet $\mathbb{Z}$ for alle heltallene. I tillegg til de naturlige tallene er også de negative og null med i denne tallmengden.

$$\mathbb{Z} = \{\cdots, -3, -2, -1, 0, 1, 2, 3, \cdots\}$$

I Python er det sånn at datatypen til en variabel bestemmes ut fra verdien vi tilordner den. Slik er det ikke alle programmeringsspråk. Fordelen er at det gjør det enklere for oss å skrive kode.

Når vi skriver kommandoen `a = 13` er verdien som tilordnes et heltall. Python bruker da datatypen heltall for variabelen `a` uten at vi må skrive noe mer. I andre språk er det vanlig at vi må si fra hvilken datatype variabelen skal ha.

Desimaltall

Desimaltall er enkle å skrive for oss, men mer problematisk å få lagra i ei datamaskin. Måten en lagrer desimaltall på gjør at de kalles *flyttall* i den her sammenhengen. I Python kalles denne datatypen for `float`.

Skriver vi nå `b = 13.0` gir det beskjed om at vi vil legge inn et desimaltall i variabelen `b` slik at den automatisk vil være av typen `float`.

Her er et program som legger inn et flyttall i en variabel og tilordner en ny variabel som kvadratet av den verdien

```
1 a = 1234567890.123456789
2 b = a*a
3 print(b)
```

Programkode 3.4: Store verdier

Programmet gir dette resultatet

```
1.5241578753238835e+18
```

Blir flyttallet stort nok vil det skrives på standardform

$$a \cdot 10^n, \qquad a \in [1, 10\rangle$$

Det betyr at tallet a skal være fra og med 1 og opp til 10 og at det skal multiplisers med en tierpotens. Kalkulatorer og datamaskiner skriver dette på sin egen måte slik at $\cdot 10^n$ blir til `en` eller `En`. Det betyr at resultatet i programkode 3.4 skal skrives slik i matematisk språkdrakt

$$1.5241578753238835 \cdot 10^{18}$$

hvor 10^{18} er skrevet som `e+18`.

Tekststrenger

Vi har sett at variabler også kan inneholde tekst. Navnet på den datatypen er *streng*, eller på engelsk *string*. I Python benyttes `str` som en forkortelse. Her er et eksempel på hvordan vi legger inn en tekststreng i en variabel

```
d = "dette er en streng"
```

Vi markerer at dette er en streng ved å skrive mellom to anførseltegn. Vi kan også benytte enkle anførseltegn, men det kan være greit å holde seg til en måte å gjøre det på.

Vi kan også utføre operasjoner med tekststrenger. Det skal vi se mere på seinere, men en liten smakebit er å sette sammen strenger (eng. *concatenate*). Operatoren for det er plusstegnet.

```
1 h = "Velkommen"
2 navn = "Olga"
3 hilsen = h + ", " + navn + "!"
4 print(hilsen)
```

Programkode 3.5: Tekststrenger

```
Velkommen, Olga!
```

De to variablene `h` og `Olga` slås sammen med tegnene `,` og `!` og tilordnes variabelen `h`. Til slutt får vi skrevet ut en ny tekststreng.

3.3 Konvertering mellom datatyper

Vi kan gjøre om en datatype til en annen ved å konvertere datatypen. Det er ikke alle slike konverteringer som går, men er noen eksempler:

```
1 var1 =  int(7.324)  # var1 settes til heltallet 7
2 var2 =  float(3)    # var2 settes til desimaltallet 3.0
3 var3 =  str(7.324)  # var3 settes til tekststrengen "7.324"
4 print(var1)
```

```
print(var2)
print(var3)
```

Programkode 3.6: Konvertering

Programmet gir dette resultatet

```
7
3.0
7.324
```

Kommandoen `int(7.324)` gjør om det som står inne i parentesen til et heltall og `float(3)` gjør om heltallet 3 til desimaltallet 3.0. Vi får gjort om desimaltallet 7.324 til en tekststreng ved kommandoen `str(7.324)`. For oss ser det som blir skrevet ut som et desimaltall, men bare prøv å bruk `var3` som et desimaltall ved å skrive f. eks. `var3*2`. Da får vi ei feilmelding som sier fra om at dette går ikke.

Slike konverteringer kan være nyttige og benyttes ofte for å sikre at datatypen blir den vi ønsker.

Vi kan også be om å få skrevet ut datatypen til variablene med kommandoen `type`.

```
a =  int(7.9)
b =  float(7)
c = 7.9
d = 7
print(a, b, c, d)
print( type(a),  type(b),  type(c),  type(d))
```

Programkode 3.7: Hvilken datatype?

```
7 7.0 7.9 7
<class 'int'> <class 'float'> <class 'float'> <class 'int'>
```

3.4 Input

Når vi programmerer er vi ofte interessert i å kommunisere med en bruker. Vi ønsker å få skrevet inn data, få gjort et eller annet, og sende det ut på skjermen. Vi har allerede brukt kommandoen `print` og fått skrevet ut og vi har sett hvordan vi kan utføre flere operasjoner. Nå skal vi se på kommandoen `input` som gjør det mulig for en bruker å skrive inn verdier.

Vi kan se på et eksempel som beregner prisen en kunde må betale for jordbær når prisen per kurv og antall kurver oppgis. Vi kunne skrive koden slik:

```
navn = "Ola"
pris = 55.60
antall = 5
totalpris = pris*antall
print(navn, "du må betale ", totalpris, "kr.")
```

Programkode 3.8: Jordbær

Resultatet blir

```
Ola du må betale 278.0 kr.
```

Nå vil vi gjøre om programmet til at en bruker gir oss navnet og de andre verdiene. Det gjør vi med `input`

```
1 navn =  input("Hva heter du?")
2 pris =  float( input("Hva er prisen på ei kurv med jordbær?"))
3 antall =  int( input("Hvor mange kurver kjøpte du?"))
4 totalpris = pris*antall
5 print(navn, "du må betale ", totalpris, "kr.")
```

Programkode 3.9: Mere jordbær

Nå har vi fått et program hvor vi henter opplysninger fra brukeren. Kommandoen `navn = input("Hva heter du?")` gjør at tekststrengen `Hva heter du?` blir skrevet til skjermen og programmet venter på hva som blir skrevet inn. Etter at noe blir skrevet, og brukeren avslutter med retur-tasten, fortsetter programmet til neste linje.

Når vi vil at brukeren skal skrive inn `antall` og `pris` sørger vi for at det blir korrekt datatype ved at vi setter hele `input`-kommandoen inne i kommandoer som gjør om til den korrekte datatypen. Hvis ikke blir det som skrives inn til tekststrenger og det kan vi ikke bruke til utregninger. Vi får et flyttall med kommandoen `float()` og heltall med `int()`.

Til slutt skrives variablene ut sammen med noen tekststrenger som passer.

Et tips når du programmerer er å starte med å tilordne verdier til variablene som i programkode 3.8. Da slipper vi å måtte svare på alle spørsmålene for å teste koden. Når alt fungerer som det skal, gjør vi om til å la brukeren legge inn verdier.

3.5 Introduksjon

Lister er en nyttig datatype i Python. Ordet liste kjenner vi igjen fra f. eks. handlelister eller ønskelister. Lister kan inneholde flere elementer og vi må heller ikke bestemme oss for hvor mange elementer det skal være. Det er bare å legge til etter behov. Ei liste oppretter vi ganske enkelt ved å gi den et navn og så skrive elementene i klammeparenteser. Etter å ha gjort det kan vi utføre operasjoner med listene, f. eks. legge sammen to lister:

```
>>> a = [1,2,3,4]
>>> min_liste_av_noen_partall = [2,4,6,8,10]
>>> a + min_liste_av_noen_partall
[1, 2, 3, 4, 2, 4, 6, 8, 10]
```

Nå skal vi se mer på hva lister er og hvordan de kan benyttes.

3.6 Lister – hva er det?

Vi starter med ei liste over plantenavn hvor hvert element er en tekststreng:

```
planter = ["løvetann","blåveis","hvitveis","rødkløver"]
```

Ønsker vi å se innholdet i lista kan vi gjøre det med kommandoen `print`

```
1 planter = ["løvetann","blåveis","hvitveis","rødkløver"]
2 print(planter)
```

Programkode 3.10: Utskrift

Vi har laget oss en programmeringsvariabel av typen liste med plantenavn. En slik samling med elementer er noe vi ofte får bruk siden vi har kommandoer og operasjoner som gjør det mulig å få gjort noe med listene eller elementer i ei liste.

3.7 Operasjoner med lister

Legge til noe i ei liste

Noen av poenget med listene er at vi kan legge til, eller trekke fra, elementer. For å legge til ett element bruker vi kommandoen `append` etter å ha oppgitt navnet på lista. Vi må også ta med hva vi ønsker å legge til. La oss si at vi ønsker å legge til *hestehov* i lista over plantenavn. Da må legge til tekststrengen `"hestehov"` på slutten av lista vår.

```
1 planter = ["løvetann","blåveis","hvitveis","rødkløver"]
2 planter.append("hestehov")
3 print(planter)
```

Programkode 3.11: Append

Det som skjer er at det legges til etter de andre tekststrengene.

Slå sammen lister

Vi kan slå sammen lister ved bruk av operatoren `+`.

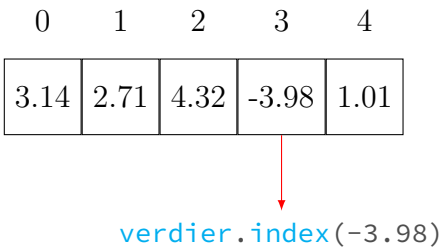
```
>>> a = [1, 2.71, "hallo"]
>>> b = [8, "hei",3.14]
>>> a + b
[1, 2.71, 'hallo', 8, 'hei', 3.14]
```

Legg også merke til at elementene i ei liste ikke må være av samme datatype.

Indeksering

Da har vi sett at vi kan legge til på slutten av lista. Her havnet «hesthov» som det femte elementet i lista, men som vi kan se ellers i programmeringsverdenen starter ikke tellinga med én. Det benyttes en nullindeksering.

Vi lager denne lista `verdier = [3.14,2.71,4.32,-3.98,1.01]`. Indeksene starter med null



Vi kan finne indeksen til et element ved å bruke kommandoen `index`. Her er et eksempel

```
>>> verdier = [3.14,2.71,4.32,-3.98,1.01]
>>> verdier.index(-3.98)
3
```

Kommandoen leter opp elementet og gir oss indeksen tilbake. Prøver vi med noe som ikke er i lista får vi ei feilmelding

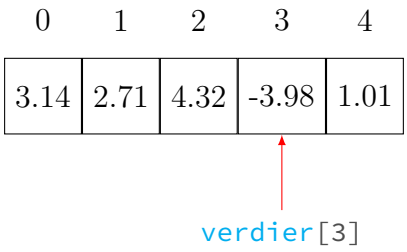
```
>>> verdier.index(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 4 is not in  list
```

Indekseringen gjør at vi kan få tilgang til hvert element i lista og det behovet oppstår når vi programmerer.

Hente ut deler av lista

Vi kan angi hvilke elementer vi ønsker å skrive ut fra ei liste ved å benytte indeksene. Vi kan benytte indeksen for å få skrevet ut det som er på den plassen eller vi kan angi flere indekser. Her er et eksempel.

```
>>> verdier = [3.14,2.71,4.32,-3.98,1.01]
>>> verdier[3]
-3.98
>>> verdier[2:4]
[4.32, -3.98]
>>> verdier[2:]
[4.32, -3.98, 1.01]
>>> verdier[:2]
[3.14, 2.71]
```



En forklaring til kommandoene

`verdier[3]`	gir elementet som er indeksert på plass tre.
`verdier[2:4]`	gir elementene **fra og med** indeks to og **til** fire.
`verdier[2:]`	gir elementene **fra og med** to og resten av lista.
`verdier[:2]`	gir elementene **fra** starten og opp **til** indeks to.

Bruk av indekser kan være lurt når vi har to lister med tilhørende elementer. La oss si at vi registrer navn og mobilnummer til hver person. Da kan det løses ved å opprette to lister: ei med navnene og ei med nummerene. Har vi navn og nummer på riktig plass løser indeksene jobben med å skrive ut både navnet og nummeret:

```
>>> navn = ["Ole", "Kari","Truls"]
>>> nummer = [9225434, 4567476, 5564489]
>>> print(navn[2], nummer[2])
Truls 5564489
```

Her har starten på en database!

Fjerne fra lista

For å fjerne et element fra lista er det bare å si fra hvilket element og bruke kommandoen `remove`

```
>>> verdier = [3.14,2.71,4.32,-3.98,1.01]
>>> verdier.remove(2.71)
>>> verdier
[3.14, 4.32, -3.98, 1.01]
```

Her kan vi legge merke til at elementet fjernes uten at rekkefølgen på de andre forandres.

Andre operasjoner

Sortering av listene kan vi gjøre med kommandoen `sort`. Bruker vi den på lista over plantenavn vil resultatet bli ei alfabetisk sortert liste over navnene.

```
>>> planter.sort()
>>> planter
['blåveis', 'hestehov', 'hvitveis', 'løvetann', 'rødkløver']
```

Antall elementer i lista finner vi ved `len`.

```
>>> verdier = [3.14,2.71,4.32,-3.98,1.01]
>>>  len(verdier)
5
```

Summen av elementene i ei liste med tall er gitt av kommandoen `sum`

```
>>> a = [1,2,3,4]
>>>  sum(a)
10
```

Antall forekomster Vi kan også finne ut hvor mange elementer som har samme verdi i ei liste. Da benytter vi kommandoen `count()`.

```
>>> min_liste = [1,1,2,3,4,4,5,5,5,6,7,7,8]
>>> min_liste.count(5)
3
>>> min_liste.count(7)
2
```

Her får vi at det er tre forekomster av 5 og to element som har verdien 7.

Største og minste verdi finne vi med kommandoene `max` og `min`.

```
>>>  max(min_liste)
8
>>>  min(min_liste)
1
```

En oversikt

Her er en oversikt over noen kommandoer som kan benyttes på lister. For flere kan det lønne seg å søke på nettet.

Listekommandoer

Kommando	Forklaring
`append()`	Legger til et element sist i lista
`clear()`	Sletter alle elementene
`copy()`	Kopierer lista
`count()`	Gir antall elementer av en gitt verdi
`index()`	Gir indeksen til den første forekomsten til en gitt verdi
`insert()`	Setter inn et element i en gitt posisjon
`pop()`	Sletter et element i en gitt posisjon
`remove()`	Sletter et element med en gitt verdi
`reverse()`	Reverserer rekkefølgen i lista
`sort()`	Sorterer lista

Tabell 3.2: Listekommandoer

Noen oppgaver

Oppgave 9

Hva gjør dette programmet?

```
1 planter = ["løvetann","blåveis","hvitveis","rødkløver"]
2 planter.append("hestehov")
3 print(planter)
4 antall =  len(planter)
5 print(antall)
```

Oppgave 10

Hva gjør dette programmet?

```
1  planter = ["løvetann","blåveis","hvitveis","rødkløver"]
2  planter.append("hestehov")
3  print(planter)
4  antall =  len(planter)
5  print(antall)
6  planter.sort()
7  print(planter)
```

Oppgave 11

Hva gjør dette programmet?

```
1  verdier = [3.14,2.71,4.32,-3.98,1.01]
2  a = verdier[2]
3  print(a)
4  b = verdier[2:4]
5  print(b)
6  c = verdier[3:]
7  print(c)
8  d = verdier[:2]
9  print(d)
10 print(a,"\n",b,"\n",c,"\n",d,"\n")
11 print( max(verdier))
12 print( min(verdier))
```

3.8 Mer om operasjoner på tekststrenger

Vi har sett at tekststrenger kan være variabler. Et eksempel var at vi deklarerte en variabel på denne måten

```
d = "dette er en streng"
```

Vi får da en streng- variabel av type `str`.

I Python kan vi utføre en rekke operasjoner på variabler av typen streng.

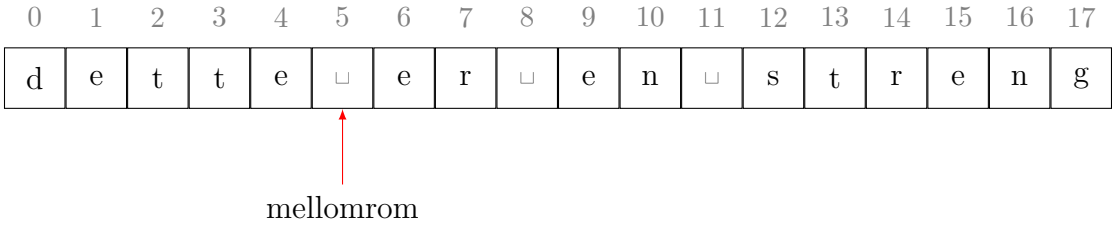
Sammensetning av tekstvariabler er kanskje den mest vanlig operasjonen. La oss se på eksempelet i programkode 3.12. Her er det flere tekstvariabler som slås sammen med operatoren `+`. Til slutt tilordnes den nye tekststrengen til en variabel og skrives ut.

```
1 a = "dette"
2 b = "er"
3 c = "en"
4 d = "streng"
5 s = a + " " + b + " " + c + " " + d
6 print(s)
```

Programkode 3.12: Slå sammen tekst

Resultatet blir at vi får skrevet ut den sammenslåtte teksten: `dette er en streng`.

Hvordan lagres en streng? Det må vi se nærmere på før vi går videre. Tekststrenger lagres tegn for tegn i ei liste. Hvert tegn har en egen indeks. Hvis vi fortsetter med tilordningen `d = "dette er en streng"` så vil den lagres på denne måten



Hvert tegn lagres fortløpende, akkurat som i ei liste, og hvert tegn indekseres. Legg merke til at her, som ellers i programmeringsverdenen, så indekseres det fra null. Legg også merke til at mellomrom, som alle andre tegn, plasseres i denne lista. Når vi ønsker å vise et mellomrom er det vanlig å benytte tegnet ␣.

Lengden av en tekststreng finner vi med kommandoen `len(<streng>)`. Skriver vi kommandoen `lengde = len(d)` vil variabelen `lengde` inneholde antall tegn som strengen `d` består av. Programkode 3.13 hvordan vi kan benytte kommandoen.

Ett tegn er det mulig å hente ut fra tekststrengen. Husk at den ble lagra som ei liste med alle tegnene. Ønsker vi å hente ut bare ett tegn kan vi gjøre det med å si fra om hvilken indeks. Skriver vi `d[4]` hentes det tegnet som er indeksert som nummer 4, dvs det femte tegnet slik vi regner.

Deler av strengen kan vi også hente ut fra lista over alle tegnene. Kommandoen som gjør det er en utvidelse av den vi benyttet for å hente ett tegn:

```
streng[startindeks: sluttindeks: steg]
```

Start- og sluttindeks er indeksverdien vi ønsker å starte og slutte med. Kommandoen `d[0:5]` gir `dette` som resultat. Alle tegnene opp til indeks 5 blir med. Mellomrommet som har indeks 5 blir ikke med.

Sier vi fra om hvor lange steg det skal være mellom det som hentes ut får det konsekvenser. Kommandoen `print(d[0:5:2])` gir `dte`. Hva skjer? Jo, intervallet er det samme, men nå blir ikke alle tegnene med. Bare hvert andre tegn hentes fra lista.

Sjekke forekomsten . Vi kan sjekke om en streng er inne i en annen ved å skrive

```
<streng> in <streng>
```

Fins en streng inne i en streng Det kan vi sjekke med `"en" in d` gir enten `True` eller `False` – sann eller usann.

```
1 d = "dette er en streng"
2 lengde =  len(d)
3 print("Denne strengen består av ", lengde, " tegn")
4 print("Det femte tegnet er: ",d[4])
5 print("Det siste tegnet er: ",d[lengde-1])
6 print("Her er en del av strengen: ",d[1:8])
7 print(d[0:5:2])
8 print("en" in d)
```

Programkode 3.13: Tekststrenger

```
Denne strengen består av 18 tegn
Det femte tegnet er: e
Det siste tegnet er: g
Her er en del av strengen: ette er
dte
True
```

Oppgave 12

Eksperimenter med de forskjellige kommandoene.

Oppgave 13

Lag et program som teller hvor mange ganger en bestemt bokstav forekommer i en tekst.

3.9 Bruk av anførseltegn

Vi markerer at dette er en streng ved å skrive mellom to anførseltegn. Vi kan også benytte enkle anførseltegn, som i denne kommandoen `d = 'dette␣er␣en␣streng'`. Det er bare i spesielle

tilfeller at det utgjør en forskjell. Ønsker vi å få skrevet ut strengen: `Det er noe "alle" vet!` kan vi velge å skrive strengen med enkle anførseltegn for å unngå feilmelding:

```
print('Det er noe "alle" vet!')
```

Hvis vi vil bruke doble går det også, men da må vi si fra om at vi mener doble anførseltegn inne mellom anførseltegnene

```
print("Det er noe \"alle\" vet!")
```

Apostrofer krever også en spesiell håndtering, men på norsk er jo det enkelt å unngå dem. Tekststrengen `NRK's kontor` krever at vi enten skriver `print("NRK's kontor")` eller `print('NRK\'s kontor')`.

Det er bare i sjeldne tilfeller vi støter på disse problemene, men oppstår det ei feilmelding kan det være greit å være klar over at det kan være her problemet ligger. Det kan også hende at du støter på forskjeller i ulike versjoner av både Python og programmet hvor du skriver kode.

4 Programmeringsteknikker

4.1 Hvordan skrive kode?

Når en skal skrive et program er det viktig med en god struktur. Med en gang koden blir mer omfattende kan det være greit å starte med papir og blyant for så å kladde seg fram til en god løsning. Da er det greit med to hjelpemidler: flytskjema og pseudokode.

4.2 Pseudokode

En pseudokode er noe midt mellom programkode og vanlig språk: En strukturert framstilling av programmet uten å tenke for mye på syntaksen som kreves for at det skal kunne fungere. På den måten får vi fram strukturen i koden samtidig som vi slipper å tenke for mye på hvilke språkspesifikke kommandoer som må benyttes. Det tar vi til slutt når pseudokoden oversettes til programmeringsspråket vi benytter. Når vi benytter Python er den prosessen ganske grei. Python likner mer på pseudokode enn hva tilfellet er med andre språk.

La oss se på et eksempel hvor vi skal prøve å løse oppgaven under

Oppgave 14

Lag et program hvor det skrives inn to ulike tall. Programmet skal vise hvilket tall som er størst.

Før vi går i gang med programmeringen kan det være lurt å skrive koden uten å tenke for mye på syntaks og hvordan kommandoene skal skrives i Python.

```
skriv inn tall a
skriv inn tall b
hvis a > b
        skriv ut: tall a er størst
eller
        skriv ut: tall b er størst
```

Slik kan det skrives, men det kan være lurt å legge seg litt nærmere syntaksen i Python med noe som dette:

```
input tall a
input tall b
if a > b
        output tall a er størst
else if a = b
                output tallene er like store
        else
                output tall b er størst
```

Eksemplet er banalt. Her er det ingen grunn til å pseudokode siden Pythonkoden er så nær, men det viser hvordan en kan gå fram uten å tenke for mye på hvordan det må skrives. I mer komplekse situasjoner kan pseudokode være nyttig.

Skulle vi løst oppgaven i Python, og sett bort fra at det må være to ulike, ville vi endt opp med noe slikt:

```
1 a =  int( input("Tall a? "))
2 b =  int( input("Tall b? "))
3 if a > b:
4     print("Tall a er størst")
5 elif a == b:
6     print("Tallene er like store")
7 else:
8     print("Tall b er størst")
```

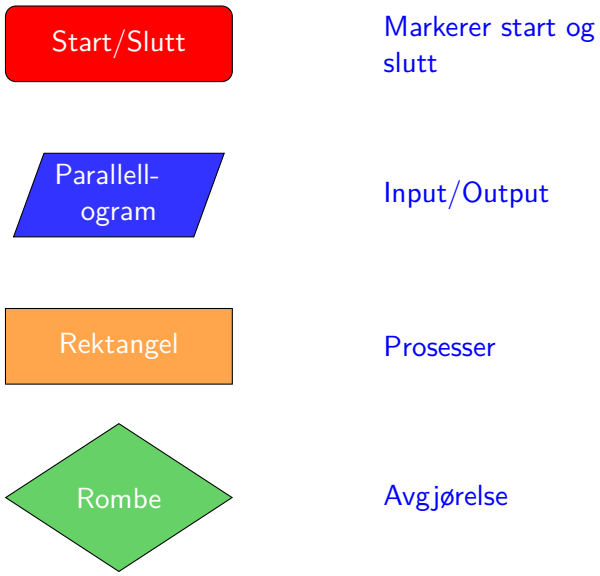
Programkode 4.1: Eksempel

Vi skal se nærmere på hva som skjer i program som dette seinere.

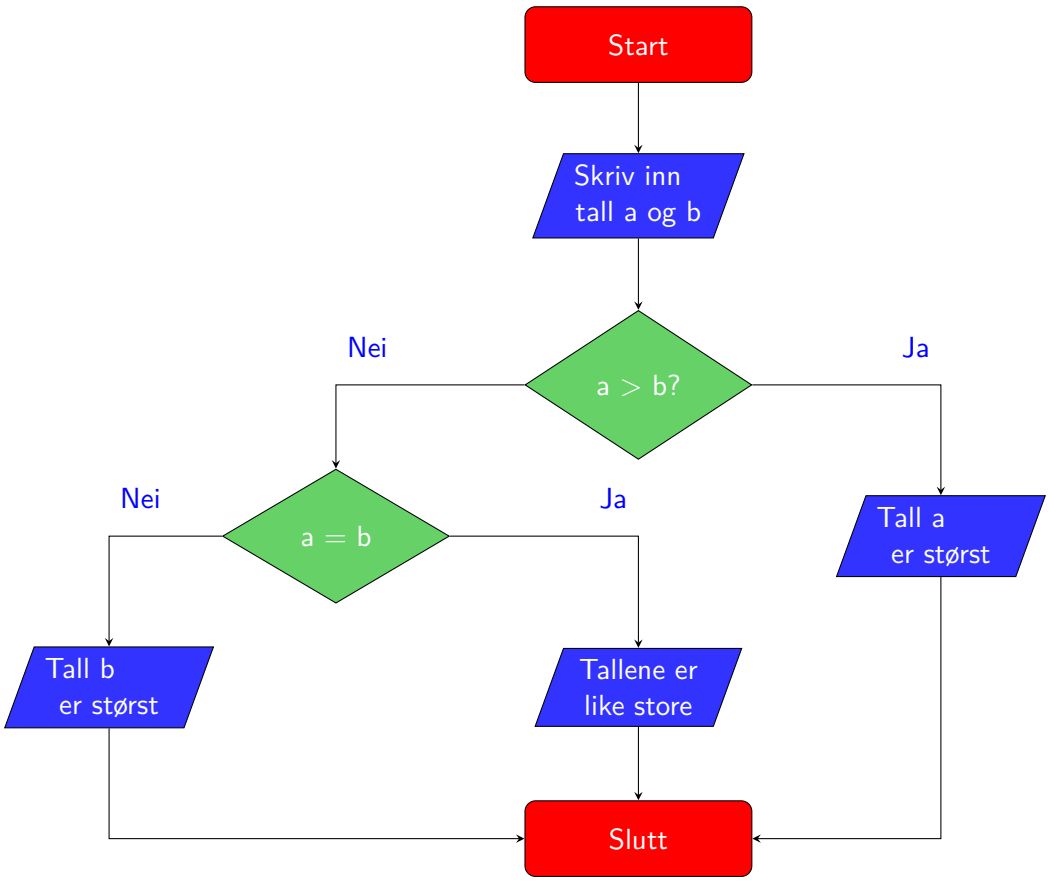
4.3 Flytskjema

Et flytskjema er også et hjelpemiddel for å få skrevet koden.

Forklaring flytskjema



Flytskjema for eksemplet


Start
Skriv inn
tall a og b
a > b?
Nei
Ja
a = b
Nei
Ja
Tall a
er størst
Tall b
er størst
Tallene er
like store
Slutt

5 Vilkår

5.1 Hvis-test

Når vi programmerer oppstår det ofte et behov for å ta en avgjørelse basert på et vilkår[1]. Vi ønsker at noe skal skje *hvis* et vilkår er stemmer eller ikke. Til slikt har vi hvis-tester. Vi se et typisk eksempel i programkode 5.1

```
1 alder =  int( input("Hvor mange år er du? "))
2 if alder > 16:
3     print("Du er voksen!")
```

Programkode 5.1: En enkel hvis-test

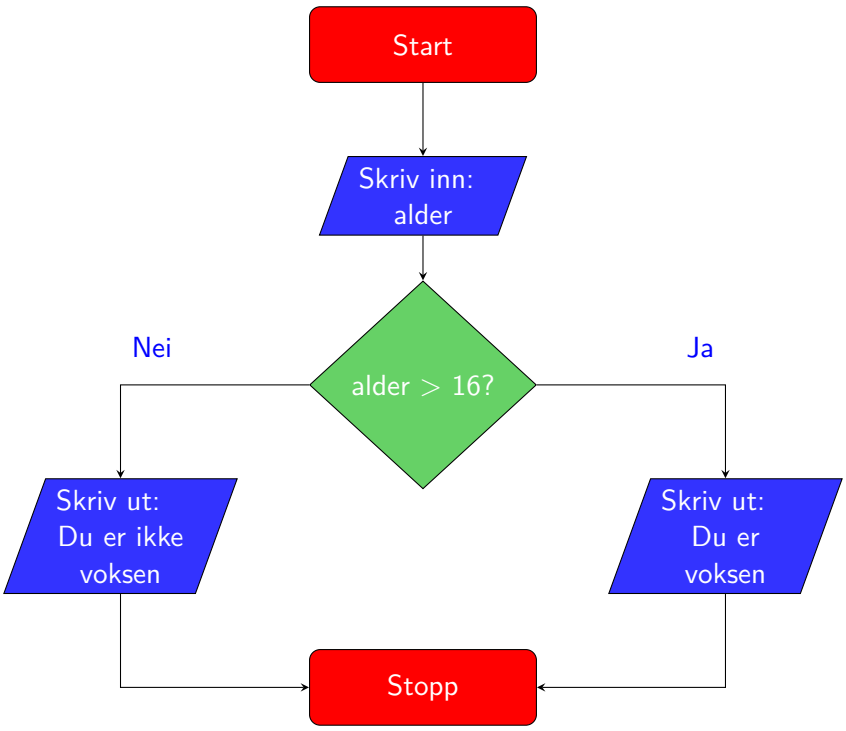
Her sjekkes det om alderen som ble skrevet inn er et tall større enn 16. Hvis det stemmer skal en tekststreng skrives ut. Hvis det ikke stemmer skjer det ikke noe som helst. For også å få skrevet ut noe når det ikke stemmer kan vi benytte kommandoen `else`. Da vil noe bli utført både om det stemmer og om det ikke gjør det.

```
1 alder =  int( input("Hvor mange år er du? "))
2 if alder > 16:
3     print("Du er voksen")
4  else:
5     print("Du er ikke voksen")
```

Programkode 5.2: En hvis-ellers-test

Dette var enkle eksempler. Med flere slike hvis-tester, ofte satt inn i hverandre, kan det bli uoversiktlig. Da er flytskjema til stor hjelp. Figur 5.1 er et som viser avgjørelsene som blir tatt i det forrige eksemplet.

[1] betingelse er et annet ord for det samme



Figur 5.1: Flyskjema med en hvis-test

Følger vi pilene i flytskjemaet får et helhetlig bilde på hva som skjer i koden. Vi slipper også å tenke på programsyntaks eller programmeringsspråk.

La oss prøve et litt mer avansert eksempel hvor flere hvis-tester er satt inn i hverandre. Vi sier at hvis-testene er *nøstet* når det kommer en ny test inn i en annen. Her er et programeksempel som ber om å få oppgitt en alder og så skriver ut forskjellige beskjeder basert på alderen.

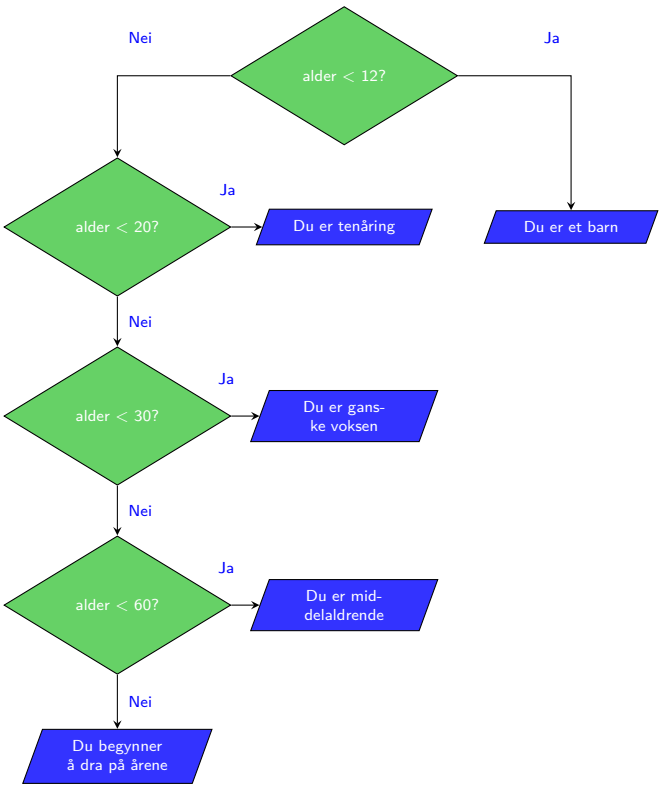
```
1  alder =  int( input("Hvor mange år er du? "))
2  if alder < 12:
3      print("Du er et barn")
4  elif alder < 20:
5      print("Du er tenåring")
6  elif alder < 30:
7      print("Du er ganske voksen")
8  elif alder < 60:
9      print("Du er middelaldrende")
10 else:
11     print("Du begynner å dra på årene")
```

Programkode 5.3: Hvis-tester som er nøstet

Programkode 5.3 kan bli uoversiktlig. Her ligger testene inne i hverandre med kommandoene `if [] elif []`. Det er en kortform for å skrive `if` og så `else` med en påfølgende hvis-test. Syntaksen er

```
1  if <vilkår>:
2     <kode>
3  elif <vilkår>:
4     <kode>
```

Med mange slike nøsta hvis-setninger kommer flytskjema til sin rett. Tegner vi opp det som skjer i koden vil vi ende opp med et flytskjema som i figur 5.2 (her er start og stopp tatt bort)



Figur 5.2: Flyskjema med flere hvis-tester

5.2 Sammenlikninger og boolske variabler

Eksemplene viser at det ofte er behov for å sammenlikne størrelser når vi programmerer. I en hvis-test sjekker vi om en påstand er sann eller usann. Slike sammenlikninger kjenner vi fra matematikken. Her er noen eksempler

$$a = b, \quad a \neq b, \quad a > b, \quad a \geq b, \quad a < b, \quad a \leq b$$

Datamaskina kan raskt svare på om dette stemmer eller ikke. I Python vil svaret enten være `True` eller `False`. Dette er de to logiske konstantene som benyttes. Logiske variabler kalles også `boolske` variabler, og vi sier at de er av datatypen `boolean`.

George Boole (1815 – 1864)

Når vi tar opp logikk i programmering støter vi på navnet Boole, som i boolske uttrykk og boolske variabler. George Boole var en engelsk matematiker og filosof og er den som skapte grunnlaget for det som kalles boolsk algebra.
Les mer om Boole på Wikipedia: https://no.wikipedia.org/wiki/George_Boole

Når vi skal sammenlikne størrelser i Python må vi benytte noen operatorer. Symbolene er ikke helt ukjente, men vi må passe på at det blir skrevet korrekt. Tabellen viser hvordan Python vil at vi skal skrive det.

Legg merke til at vi benytter to likhetstegn for å undersøke om to størrelser er like hverandre.

I programkode 5.4 skrives resultatet av noen sammenlikninger ut.

```
1 a = 1
2 b = 2
```

Sammenlikninger		
Tegn	**Operasjon**	**Eksempel**
>	Større enn	a>b
<	Mindre enn	a<b
==	Er lik	a==b
>=	Større enn eller lik	a>=b
<=	Mindre enn eller lik	a<=b
!=	Ikke lik	a!=b

Tabell 5.1: Sammenlikninger

```
3 print("a > b er: ", a > b)
4 print("a < b er: ", a < b)
5 print("a == b er: ", a == b)
6 print("a != b er: ", a != b)
```

Programkode 5.4: Sammenlikninger

Kjører vi denne koden vil resultatet bli:

```
a > b er: False
a < b er: True
a == b er: False
a != b er: True
```

Python utfører sammenlikningene og finner at resultatet enten blir `True` eller `False`.

Oppgave 15

Eksperimenter med andre verdier av `a` og `b` og se om du får det resultatet du forventer.

5.3 Logiske operatorer

Vi har sett hvordan vi kan sammenlikne størrelser og få svar på sammenlikningene som sanne eller usanne. Sammenlikningene er boolske variabler, som enten er `True` eller `False`. I Python kan vi også utføre operasjoner med slike boolske variabler med det som kalles logiske operatorer. Vi sier at vi setter opp boolske uttrykk.

Det gir oss behov for en tur innom logikken og de logiske operatorene. I Python er de viktigste

Logiske operatorer	
Navn	**Python kommando**
OG	and
ELLER	or
IKKE	not

Tabell 5.2: Logiske operatorer

Vi kan se på et eksempel som benytter `and` og `or`.

```
a = 1
b = 2
c = -1

if a > 0 and b > 0 and c > 0:
    print("Alle tallene er større enn 0")
else:
    print("Minst ett av tallene er mindre enn eller lik 0")

if a > 1 or b > 1 or c > 1:
    print("Minst ett av tallene er større enn 1")
else:
    print("Alle tallen er mindre eller lik 1")
```

Programkode 5.5: Bruk av logiske operatorer

I programkode 5.5 starter vi med å legge inn tre verdier i tre variabler. I hvis-setningene benytter vi sammenlikninger satt sammen med logiske operatorer.

Oppgave 16

Skriv av koden og eksperimenter med forskjellig verdier. Prøv å forutsi resultatet før du kjører koden.

Når vi benytter logiske operatorer kan det være greit å se på en sannhetstabell. Setter vi opp verdier for to boolske variabler, `a` og `b`, kan vi få denne oversikten

Sannhetstabell

a	b	a and b	a or b
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Tabell 5.3: Sannhetstabell

Her ser vi at `a and b` bare er `True` hvis begge variablene `True`. Vi kan også legge merke til at `a or b` bare er `False` hvis begge er det.

Vi kan la Python skrive ut det samme ved å legge inn forskjellige boolske konstanter i variabler, slik som programkode 5.6 viser.

```
x = True
y = False

print("x and y er: ", x and y)
print("x or y er: ", x or y)
print("not x er: ", not x)
```

Programkode 5.6: Logiske operatorer

Her får vi også vist hvordan operatoren `not` kan benyttes som en *negasjon*. Hvis noe ikke er sant er det usant. Er det ikke usant er det sant. Resultatet blir:

```
x and y er: False
x or y er: True
not x er: False
```

Vi kan nå vende tilbake til programkode 5.5 hvor vi benyttet de logiske operatorene på forskjellig sammenlikninger. Den første hvis-setningen var: `if a > 0 and b > 0 and c > 0:` Med de opprinnelige verdiene for variablene vil vi få disse resultatene

`a > 0`	`b > 0`	`c > 0`	`a > 0 and b > 0 and c > 0`
`True`	`True`	`True`	`True`

Vilkåret i hvis-setningen oppfylles og vi får utført det som kommer i blokka etter setningen. Det som ikke er oppfylt vil utføres i `else`-delen. Legg merke til at `else` vil være negasjonen av vilkåret, nemlig: `not(a > 0 and b > 0 and c > 0)`, som bare vil være `False` i tilfellet over. For alle andre resultat av sammenlikningene vil vi få `True`.

6 Løkker

Løkker, eller sløyfer, er programkode som gjentas flere ganger. Vi kan skrive kode slik at en programblokk gjentas

- et bestemt antall ganger
- til et bestemt krav er oppfylt
- for alltid

I programmene vi skriver ønsker vi å unngå den siste varianten – at løkkene gjentas for alltid. Vi ønsker å få kode til å gjentas til noe oppfylles eller et visst antall ganger. I virkeligheten er det derimot ofte ønskelig at noe skal gjentas for alltid. Tenk bare på styringssystemer hvor noe skal sjekkes hele tida.

Vi skal se på to typer løkker og vi starter med at vi ønsker å gjenta noe et bestemt antall ganger.

6.1 For-løkker

Ei `for`-løkke kan benyttes for å få gjentatt en kode et antall ganger og med verdier vi bestemmer. Et eksempel er

```
1 for i in  range (2,10):
2     print(i)
```

Programkode 6.1: Ei enkel `for`-løkke

Det denne løkka gjør er å gjenta setningen `print(i)`. Hvordan den skal gjentas bestemmes av det som er satt opp i starten.

`for i in range (2,10):` forteller at variabelen `i` skal være et heltall i området fra 2 og opp til 10. Det som står i blokka under vil gjentas og `i` vil økes med 1 for hver gang. Kjører vi denne koden vil tallene Skriver ut alle tallene 2, 3, 4, 5, 6, 7, 8 og 9, skrives ut. Legg merke til at tallet 10 ikke skrives ut. Syntaksen er `<variabelnavn> in range (<startverdi>, <sluttverdi>)` hvor sluttverdien ikke er med.

Hvis vi ikke sier fra om annet økes `in range()` med 1, men en kan også få til å øke med andre steg, f.eks. 5. Da kan vi sette steglengden til slutt: `range(1,100,5)`

I eksemplet over så vi at en variabel var av typen heltall og vi fikk verdien til å øke med en gitt steglengde. Vi kan også benytte `for`-sløyfer hvor variabeltypen og verdiene bestemmes på en annen måte. Ofte kan det være verdier i lister, slik vi ser i programkode 6.2.

```
1 navn = ["Ola", "Kari", "Jens"]
2 for i in navn:
3     print(i)
```

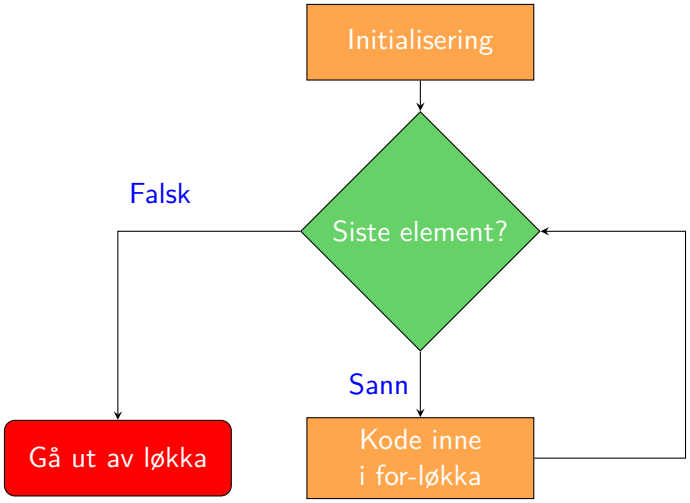
Programkode 6.2: Løkke med liste

Vi vil få skrevet ut innholdet i lista. Koden starter med at variabelen `i` får tilordna det første elementet i lista, resten av koden i blokka vil utføres, og så vil variabelen økes til neste element. Det hele gjentas helt til siste element. Vi får denne utskrifta:

```
Ola
Kari
Jens
```

Flytskjema

Flytskjema kan også være nyttige for å vise løkker. En skjematisk framstilling av ei `for`-løkke blir slik:



Koden vil gjentas helt helt til vi kommer til siste element. Med `for`-løkker kan vi gjenta kode og vi har kontroll med hvor mange ganger koden gjentas samtidig som vi kan kontrollere hver verdi som tilordnes.

6.2 While

Vi har også en annen type løkker: `while`-løkker. Ei slik løkke vil gjentas så lenge et visst vilkår er oppfylt. På norsk kan vi kalle det *så-lenge-løkke*, og den bygger vi opp på denne måten:

```
1 så lenge <vilkår>:
2     gjør det her
3
4 fortsett koden
```

Her er det eksempel:

```
1 i = 1
2 while i < 10:
3     print("Nå er i: ",i)
4     i = i+1
```

Programkode 6.3: `while`-løkke

Vilkåret er at `i < 10`. Inne i løkka skjer det noe med variabelen `i` og etter at all koden i blokka er utført sjekkes det om vilkåret er oppfylt. Resultatet blir det samme som i programkode 6.1 – tallene fra og med 1 til og med 9 skrives ut.

For å oppnå det samme kunne vi også brukt ei `for`-løkke, men i andre situasjoner er det bare `while` som gjelder. Se bare på eksemplet under.

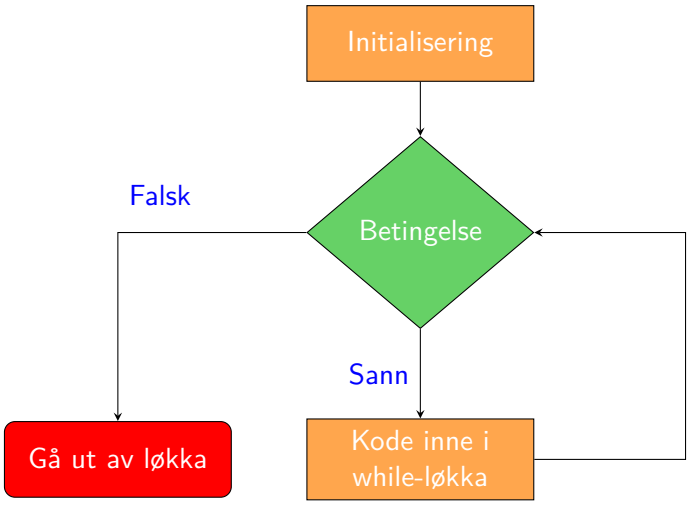
```python
ferdig = False

while not ferdig:
    print("Nå er du inne i løkka")
    svar =  input("Ønsker du å gå ut? (j/n): ")
    if svar == "j":
        ferdig = True
```

Programkode 6.4: Ei `while`-løkke til

Vi ønsker å fortsette med noe helt til brukeren velger å gå ut av løkka. For å oppnå det kan vi ikke bruke `for`. I dette eksemplet er det også brukt en boolsk variabel for å vise hvor anvendelige slike er. Alternativt kunne vi bare skrevet `while svar != "j"`, men ved å innføre variabelen `ferdig` oppnår vi mer fleksibilitet hvis vi hadde innført flere vilkår for å avslutte.

Flytskjema

Flytskjema for ei while-løkke



Oppgave 17

Lag et program som beregner hvor mange år et beløp må stå i banken for at det skal vokse til et nytt beløp. Velg renta sjøl.

7 Funksjoner

Nå skal vi se på

- hva funksjoner er

7.1 Hva er en funksjon i Python?

De fleste programmeringsspråk gir oss muligheten til å lage funksjoner, delprogram eller subrutiner – her fins det mange navn for det samme. I Python kalles slikt for en funksjon. Et eksempel kan være at vi skal skrive ut navn og adresse til noen personer. Da kan det være på sin plass med et lite delprogram som tar seg av den jobben.

```
1 # Delprogram
2 def skriv_ut(navn,sted):
3     print("Navn: ",navn, "Adresse: ", sted)
4
5 # Hovedprogram
6 skriv_ut("Ola","Sandefjord")
7 skriv_ut("Kari", "Trondheim")
```

Programkode 7.1: Delprogram

I eksemplet er det definert en pythonfunksjon[1] som har fått navnet `skriv_ut`. Navnet velges fritt, men det må ikke være noen mellomrom i det. Denne funksjonen mottar variabler og gjør noe med dem. Alt som skal gjentas mange ganger kan med fordel bli gjort med funksjoner. Det effektiviserer koden og vi får samtidig delt opp koden vår i delprogram.

Resultatet blir denne utskrifta:

```
Navn: Ola Adresse: Sandefjord
Navn: Kari Adresse: Trondheim
```

La oss nå si at vi ønsket et litt annet format på utskrifta. Da kunne vi enkelt forandre det som funksjonen gjør og fått endret alle utskriftene. La oss si til noe sånt:

```
1 def skriv_ut(navn,sted):
2     print("Jeg heter "+ navn + " og er fra "+ sted)
```

Programkode 7.2: Ny utskrift

Vi skal nå se mer på hvordan vi kan benytte funksjoner i Python

7.2 Definisjon av funksjoner

La oss si at vi har gitt en matematisk funksjon $y = f(x) = x^2 + 3$ og skal finne funksjonsverdien. I Python kan vi løse oppgaven slik: Vi legger inn en verdi for x og regner ut en y-verdi.

[1] I Python kalles dette bare en funksjon. Jeg kommer også til å bare bruke ordet «funksjon», men det er viktig å være klar over at ordbruken skiller seg fra hva en funksjon er i andre sammenhenger.

```
x = -1
y = x**2+3
print(y)
```

Programkode 7.3: Uten bruk av funksjon

Husk at skrivemåten `x**2` betyr x^2.

I første linje setter vi variabelen `x` til -1, så får vi regna ut y til å bli fire – og til slutt skrives verdien av y ut. Vi får da skrevet ut fire.

Ganske greit, men lite fleksibelt. Hva om vi ønsker å finne flere funksjonsverdier? Da må vi gjenta det hele. For mer fleksibilitet kan vi definere funksjoner i python.

Før vi går i gang må vi se på hvordan vi kan definere funksjoner i Python. Husk at funksjoner i programmering ikke nødvendigvis er det samme som i matematikken. Funksjoner slik vi definerer dem i skolematematikken mottar en innverdi og gir en entydig utverdi. Vi kan lage funksjoner i Python som har egenskapene, men vi kan også lage funksjoner som ikke faller inn under definisjonen vi kjenner fra matematikken. I det første eksemplet så vi en pythonfunksjon som ikke har egenskapene som kreves av en matematisk funksjon. Pythonfunksjonen i det tilfellet er et delprogram eller subrutine.

Nå kan vi prøve å definere en pythonfunksjon som oppfører seg som en matematisk funksjon. Programkode 7.4 viser et slikt eksempel. Den første linja `def f(x):` forteller at vi ønsker å definere en funksjon som er kalt `f`. Funksjonen skal motta en verdi som i funksjonen har variabelnavnet `x`. Etter kolon vil det som er rykket inn være kommandoene i funksjonen. Til slutt er det spesifisert hva som skal returneres fra funksjonen.

```
def f(x):
    y = x**2+3
    return y
#Hovedprogram
a = f(1)
print(a)
```

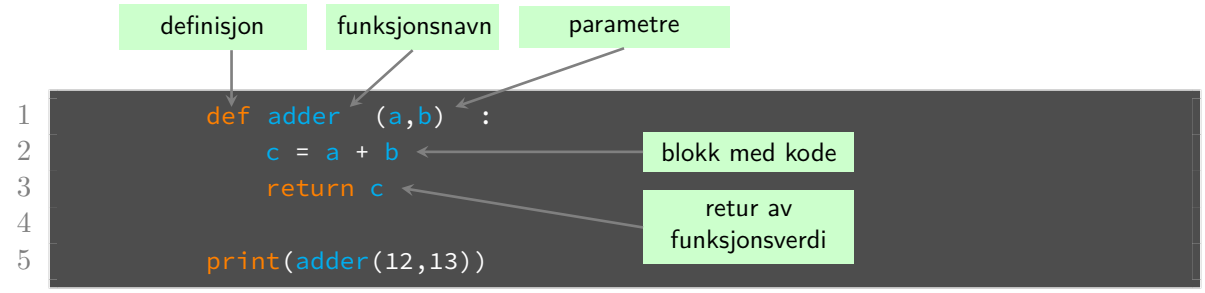
Programkode 7.4: Definisjon av en funksjon

Når vi bruker en funksjon i en kode sier vi at vi *kaller* på funksjonen. Her gjør vi det ved kommandoen `a = f(1)`. Verdien 1 overføres til variabelen `x` inne i funksjonen. I vårt tilfelle benyttes den overførte verdien i ei utregning. Til slutt returneres verdien `y` tilbake til hovedprogrammet og tilordnes verdien `a`.

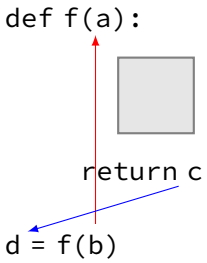
Figur 7.1 gir forklaring til et eksempel hvor det er definert en funksjon som legger sammen to tall. Funksjonen har fått navnet `adder`. I hovedprogrammet finner vi kommandoen `print(adder(12,13))` hvor vi ber om å få skrevet ut resultatet av verdien vi får tilbake fra funksjonen. Funksjonen er definert med et funksjonsnavn og *parametre*. Her er det to parametre som har fått navnene `a` og `b`. Etter at vi kaller på funksjonen med med `adder(12,13)` vil `a` tilordnes verdien 12 og `b` tilordnes 13. Kodeblokken i funksjonen legger de to sammen og sender verdien 25 tilbake.

Mer skjematisk kan det framstilles som i figur 7.2 hvor pilene viser overføringen av verdier til parametrene.

Her blir det kanskje mange piler, begrep og teoretisk. Litt eksperimentering kan være et godt tips for å rydde opp i teorien.



Figur 7.1: Definisjon og bruk av funksjon i Python



Figur 7.2: Parameteroverføring

Eksemplet har med en del variabler for å vise verdiene tydelig. Kommandoen `return` kan inneholde en utregning, slik at vi kunne skrevet `return a + b` og gjort funksjonen `adder` ei linje kortere. Programkode 7.5 viser hvordan vi kan returnere verdier som regnes ut.

```
1 def f(x):
2     return x**2+3
3
4 def g(x):
5     return x*4
6
7 print(g(3) - f(1))
```

Programkode 7.5: To funksjoner

For mange er kort kode, og bruk av få variabler, viktig, men for vår del er det en smakssak.

Funksjoner i Python

Syntaksen for funksjoner

```
1 def funksjonsnavn (parametre):
2     # kommandoblokk
3     return returvariabler
```

parametrene er ei liste med de nødvendige parametrene. Det må ikke være noen parametre eller returvariabler.

7.3 Lokale og globale variabler

Variabler kan være tilgjengelig uansett hvor i programkoden vi benytter dem eller bare være tilgjengelige innafor en funksjon. De to typene kalles *globale* og *lokale*.

Programkode 7.6 viser at det defineres en global variabel `a = 3`. Denne variabelen er tilgjengelig for alle deler av koden – global viser til akkurat det. Inne i blokka til funksjonen er det også definert en variabel med samme navn som den globale, men den er satt til en annen verdi.

```
1 a = 3
2
3 def en_funksjon():
4     a = 14
5     print("inne i funksjonen",a)
6
7 # hovedprogram
8 en_funksjon()
9 print("utenfor funksjonen",a)
```

Programkode 7.6: Lokale og globale variabler

Hva skjer? Kjører vi programmet ender vi opp med dette resultatet:

```
inne i funksjonen 14
utenfor funksjonen 3
```

Utskrifta viser at tilordninga `a = 14` ikke påvirker verdien til den globale variabelen `a`. Det som blir gjort inne i funksjonen påvirker ikke resten. Den variabelen er lokal.

Globale variabler er tilgjengelige for alle deler av koden. I programkode 7.8 benyttes verdien av den globale variabelen `a` inne i funksjonen. Her er `b` en lokal variabel som forblir ukjent for resten av koden.

```
1 a = 3
2 def en_funksjon():
3     b = 2*a
4     print("b: ",b)
5
6 # hovedprogram
7 en_funksjon()
8 print("a: ",a)
```

Programkode 7.7: Lokale og globale variabler

Dette programmet gir

```
b: 6
a: 3
```

Et forsøk på å skrive ut verdien av `b` i hovedprogrammet med kommandoen `print(b)` gir meg feilmeldinga: `NameError: name 'b' is not defined` og det må jeg si meg enig i siden `b` er lokal.

Det er kanskje ikke ofte vi får bruk for det, men vi kan si fra om at vi ønsker å definere en global variabel inne i koden til en funksjon. Det gjør vi ved å sette `global` foran variabelnavnet.

```
1 def en_funksjon():
2     global a
3     a = 21
4
5 en_funksjon()
6 print("a: ",a)
```

Programkode 7.8: Definisjon av global variabel

Det er lurt å holde styr på hva som skal være lokale og globale variabler. En god skikk er å bare bruke lokale variabler inne i funksjonen. Vi sender over variabelverdier som parametre og returnerer noe tilbake. Alt som skjer av kode inne i funksjonen bør benytte lokale variabler. En slik bruk gjør det enklere å unngå feil siden alt som skjer inne i funksjonen ikke påvirker resten av programkoden. En god regel, men det hender at vi gjør unntak.

Oppgave 18

Eksperimenter med lokale og globale variabler.

8 Bibliotek

Moduler, pakker og bibliotek er navn vi støter på når vi skal utvide funksjonaliteten til et program. Ved enkel programmering er det ikke så nøye hva vi kaller det, men for de som er mer opptatt av korrekt språkbruk kan vi ta med denne forklaringen

Moduler og pakker

Fra dokumentasjonen av Python:

Module: A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended.

Package: Packages are a way of structuring Python's module namespace by using "dotted module names".

Vi kan altså betrakte moduler som filer, og pakker som noe som strukturerer flere slike filer. Bibliotek (eng. library) er et ord som ikke er klart definert i Python, men det benyttes ofte som en referanse til samlinger av moduler, pakker og annet. Her vil jeg stort sett holde meg til ordene modul og bibliotek uten å tenke for mye på den tekniske definisjonen. Vi kan tenke på det som samlinger av programkode, definisjoner og annet som andre har laget for oss. Skal vi lage litt mer avanserte programmer får vi ofte behov for funksjonalitet som ikke er tilgjengelig uten å hente inn noe ekstra.

8.1 Hva er en modul?

For å se nærmere på hvordan vi kan utvide det som allerede fins innebygd i Python kan vi se på hvordan vi kan lage en egen modul.

I Python kan vi definere våre egne funksjoner. La oss si at det er funksjoner vi vil benytte flere ganger. Da kan vi lagre funksjonene våre i ei fil[1], la oss kalle den `minmodul.py`. Innholdet i fila finner du i programkode 8.1.

```
1 def f(x):
2     return x**2+3
3
4 def g(x):
5     return x*4
```

Programkode 8.1: minmodul.py

To funksjoner er definert, og nå kan vi importere funksjonene inn i andre program på flere måter.

Skriver vi `import minmodul` vil fila vår importeres[2]. For å kalle opp de enkelte funksjonene som er definert i modulen må vi legge til modulnavnet. I programkode 8.2 ser vi koden som må benyttes.

```
1 import minmodul
2
3 print(minmodul.g(3) - minmodul.f(1))
```

[1] Akkurat det med å lagre ei fil kan gi noen utfordringer avhengig av programmeringsmiljø. Her er det ment som en forklaring.

[2] Det krever at programmeringsmiljøet finner fila vår

Programkode 8.2: import minmodul

For å slippe å skrive lange modulnavn kan vi også importere modulen med et alias. La oss velge bokstaven m og skrive `import minmodul as m`

```
1 import minmodul as m
2
3 print(m.g(3) - m.f(1))
```

Programkode 8.3: import minmodul as m

Disse to variantene er de mest ryddige siden den importerte modulen holdes unna alle kommandoene som allerede er innebygd som standard i Python. Et ord en støter på i den sammenheng er *namespace*. Her ligger alle kommandoene som er innebygd, funksjoner vi definerer og mye annet. Måten vi har importert modulen på klusser ikke til det som ligger der fra før. Nye kommandoer holdes for seg sjøl. Ulempen er at vi må si fra at kommandoene er i en importert modul.

Vi kan importere modulene inn i *namespace*, men da bør vi være oppmerksom på det som kalles *namespace polution*. I enkel programmering er det som regel ikke noe problem, men det kan være greit å etablere en god praksis fra start. La oss allikevel se på hvordan vi kan importere moduler inn på en måte som gjør at vi slipper å skrive modulnavnet. Det gjør vi ved å skrive `from modulnavn import *`.

```
1 from minmodul import *
2
3 print(g(3) - f(1))
```

Programkode 8.4: import *

Her er hele modulen importert – stjerna sier det. Funksjonene g og f har blitt en del av de kommandoene vi kan benytte. En slik import kan altså skape litt kluss. Hva om Python allerede inneholdt en kommando som heter f eller g? Vi fyller også opp *namespace*.

En analogi som kan illustrere dette er å se på modulnavnene vi importerer som etternavn og så får med alle familiemedlemene.

Importerer vi modulene med kommandoene `import Hansen`, `import Olsen` og `import Jensen`, vil vi enkelt kunne adressere alle familiemedlemmene. En importering inn i namespace vil føre til rot siden det er flere fornavn som går igjen.

For å ikke få importert alle funksjonene i modulen kan vi fortelle hvilken funksjon vi er ute etter. La oss si at vi bare ønsker å importere funksjonen f.

```
1 from minmodul import f
2
3 print(f(1))
```

Programkode 8.5: import f

Da har vi kontroll på hva vi legger til og unngår å få med alt annet, men også denne praksisen kan komme i konflikt med andre funksjoner. Konklusjonen er at det er en god programmeringspraksis å unngå denne måten å importere på. På den annen side kan det vurderes om hva som er det beste pedagogisk. En slik import gjør det enklere å skrive koden for nybegynneren ved at det ikke må refereres til modulen når kommandoene kalles. I eksemplene som benyttes i dette kompendiet har jeg valgt å ikke bruke en slik import.

8.2 Pakker vi ofte importerer

Dette var et enkelt eksempel på en egen modul, men stort sett importerer vi moduler som andre har skrevet. Heldigvis er det mange som allerede har laget moduler med mye nyttig for oss.

8.2.1 Math

Uten å importere tilleggskommandoer er Python begrenset. Skal vi utføre matematiske operasjoner får vi ofte bruk for *Math*.

Vi importerer den og ber python om å liste opp alle funksjonene.

```
1 import math
2 print( dir(math))
```

Programkode 8.6: Liste av innholdet

Da får vi dette resultatet.

> acos, acosh, asin, asinh, atan, atan2, atanh, ceil, comb, copysign, cos, cosh, degrees, dist, e, erf, erfc, exp, expm1, fabs, factorial, floor, fmod, frexp, fsum, gamma, gcd, hypot, inf, isclose, isfinite, isinf, isnan, isqrt, ldexp, lgamma, log, log10, log1p, log2, modf, nan, perm, pi, pow, prod, radians, remainder, sin, sinh, sqrt, tan, tanh, tau, trunc

Her kan vi finne kommandoen *sqrt*, som står for square root eller kvadratrot. Å kunne finne kvadratrota av et tall uten å skrive mye kode kan vi da enkelt gjøre ved å importere math og benytte kommandoen.

Importen av `math` kan skje slik det er beskrevet tidligere. Vi kan importere både med og uten alias – og vi bare importere det vi har bruk for.

```
1 import math
2 a = math.sqrt
    (2)
3 print(a)
```

```
1 import math as m
2 a = m.sqrt(2)
3 print(a)
```

```
1 from math import sqrt
2 a = sqrt(2)
3 print(a)
```

Legg merke til at vi hvis vi velger å bare importere `sqrt`, så skriver vi `from math import sqrt` og kan benytte kommandoen direkte uten å måtte skrive navnet på modulen.

Noen viktige i Math-modulen

Math-modulen inneholder også alle trigonomteriske funksjoner som sinus, cosinus og tangens. For å finne sinus til en verdi kan vi skrive `math.sin()` og for å finne vinkelen når vi vet forholdet, `math.asin()`. Den siste står for arcus sinus. På kalkulatoren benyttes ofte $\sin^{-1}$. Tilsvarende

Metode	Beskrivelse
`math.ceil()`	Runder av verdi til opp til nærmeste heltall
`math.comb()`	Returnerer antall kombinasjoner
`math.degrees()`	Konverterer fra radianer til grader
`math.exp()`	Returnerer e opphøyd i x
`math.fabs()`	Returnerer absoluttverdien
`math.factorial()`	Returnerer fakultet av et tall
`math.floor()`	Runder av ned til nærmeste heltall
`math.gcd()`	Returnerer største felles divisor til to heltall
`math.log()`	Returnerer den naturlige logaritmen
`math.log10()`	Returnerer 10-er logaritmen
`math.perm()`	Returnerer antall permutasjoner
`math.pow()`	Returnerer verdien av x opphøyd i y
`math.sqrt()`	Returnerer kvadratrota av et tall

Tabell 8.1: Math-modulen

er det for de andre trigonometriske funksjonene. Ei liste over noen kommandoer kan du finne i tabell 8.1.

Konstantene π og e finner vi også i denne modulen som `math.pi` og `math.e`.

Et eksempel hvor vi benytter en god tilnærming til $\pi \approx 3.141592653589793$

```
1 import math as m
2 radius = 3
3 areal = m.pi*radius**2
4 omkrets = 2*m.pi*radius
5 print(areal)
6 print(omkrets)
```

Programkode 8.7: Sirkel

8.2.2 Random

Random inneholder kommandoer som gjør at vi kan få generert tilfeldige tall og en del annet. Tabell 8.2 viser noen av det vi kan få bruk for.

Metode	Beskrivelse
`random.seed()`	Initialiserer tallgeneratoren
`random.randint()`	Gir et tilfeldig heltall i et gitt intervall
`random.random()`	Gir et tilfeldig tall mellom 0 og 1
`random.choice()`	Velger et tilfeldig element fra ei liste

Tabell 8.2: Random-modulen

```
1 import random
2 print(random.randint(0,100))
3 print(random.random())
4 print(random.choice( ["rød", "blå", "hvit", "grønn"] ))
```

Programkode 8.8: Eksempler fra Random

8.2.3 PyPlot

Pyplot er en modul vi finner som en del av den større modulen Matplotlib. Vi benytter den for grafiske framstillinger som grafer, punktplott, søylediagram osv. For å importere modulen skriver vi `import matplotlib.pyplot as plt`. Da får vi importert Pyplot med aliaset plt. Vi kommer til å se mer på Pyplot når vi skal plotte data.

Les mer om Matplotlib her https://matplotlib.org

Her er en introduksjon til Pyplot: Pyplot tutorial

8.2.4 NumPy

NumPy er en forkortelse for *Numerical Python*. Modulen gir oss tilgang til å behandle matriser og arrays. I tillegg inneholder modulen en del av de samme kommandoene som vi finner i Math, f.eks. kvadratrot. Mer om NumPy kan du finne på disse sidene https://numpy.org

9 Tegne grafer

Grafer til funksjoner og punktplott er det ofte nyttig å tegne. Som regel har vi godt egna verktøy til å gjøre det både enklere, og raskere, enn med Python. I noen tilfeller kan det være greit å vite hvordan vi kan få gjort det i Python også. Her kan du lese mer om hvordan

Nå skal vi se på hvordan vi kan

- ❏ tegne grafer til funksjoner
- ❏ plotte data

Tidligere har vi sett hvordan funksjoner defineres og nå skal vi ta det i bruk når vi skal tegne grafen til en funksjon. Før vi gjør det lager vi noen punktplott.

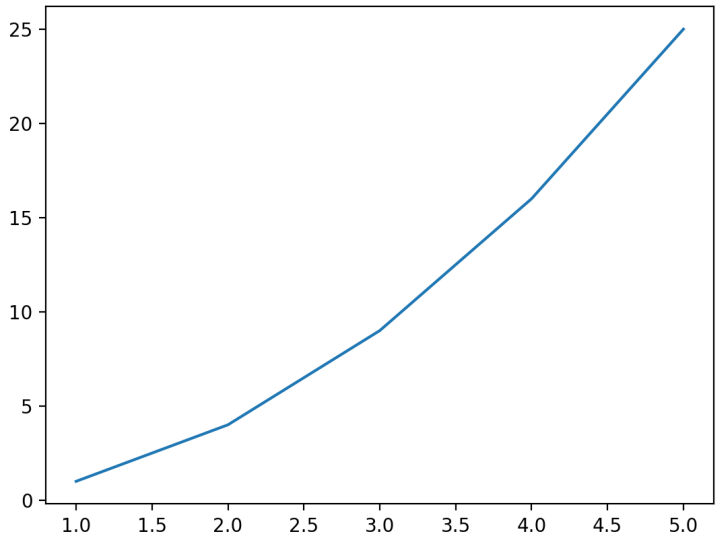
9.1 Punktplott

La oss starte med et enkelt eksempel og se på programkode 9.1. Før vi kan starte må vi legge til en del kommandoer som ikke er direkte innebygd i Python. Vi må importere det som skal til for å kunne få tegnet opp det vi ønsker. Vi importerer PyPlot med kommandoen: `import matplotlib.pyplot as plt`. Nå vil vi få mange nye kommandoer som kan benyttes med aliaset `plt`. Først må vi ha noen verdier som skal plottes. Det skaffer vi oss ved å opprette to lister med navnene x og y. Vi kan lage plottet med kommandoen `plt.plot(x,y)`: plott verdiene til de to listene x og y.

```
1  import matplotlib.pyplot as plt
2  # legger inn x-verdier og y-verdier
3  x = [1,2,3,4,5]
4  y = [1,4,9,16,25]
5
6  # plotter verdiene
7  plt.plot(x,y)
8
9  # viser plottet
10 plt.show()
```

Programkode 9.1: Et enkelt plott

Her kan det være grunn til å legge merke til at kommandoen `plt.plot(x,y)` plotter punktene, men de vises ikke. Først når vi ber om det med `plt.show()` vil vi få sett punktene i et koordinatsystem. Da bør resultatet være som i figur 9.1. Der ser vi punktene plottet og det er tegnet linjer fra det ene punktet til det andre. Verdiene langs de to aksene tar Python seg av.



Figur 9.1: Et enkelt plott

Litt pynt

Kanskje kan det gjøre seg med litt forklaring på aksene? Det kan vi få med noen kommandoer til. I programkode 9.2 er det lagt til navn på aksene og ei overskrift. Et rutenett i bakgrunnen kan også gjøre seg. Her er det lagt til med `plt.grid()`.

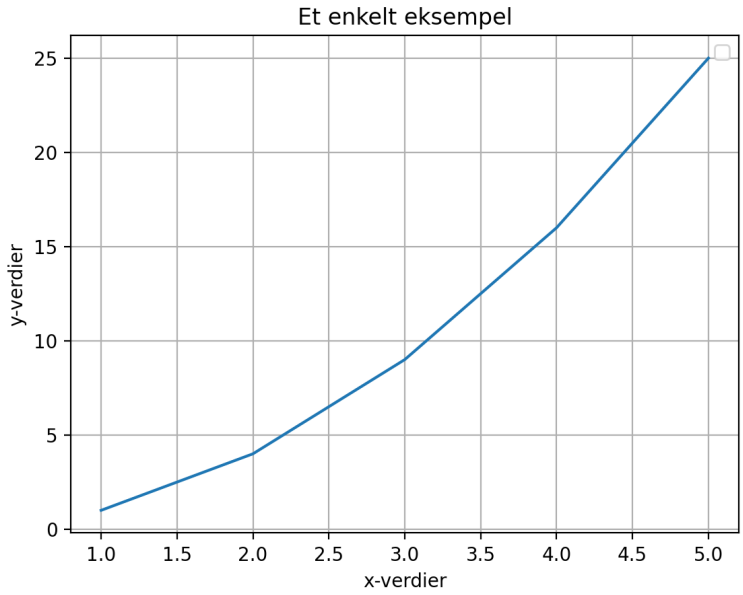
```
1  import matplotlib.pyplot as plt
2  # legger inn x-verdier og y-verdier
3  x = [1,2,3,4,5]
4  y = [1,4,9,16,25]
5
6  # plotter verdiene
7  plt.plot(x,y)
8  # tekst
9  plt.xlabel("x-verdier") # tekst på x-aksen
10 plt.ylabel("y-verdier") # tekst på y-aksen
11 plt.title("Et enkelt eksempel") # overskrift
12 plt.grid() # rutenett
13 # viser plottet
14 plt.show()
```

Programkode 9.2: Et plott med litt pynt

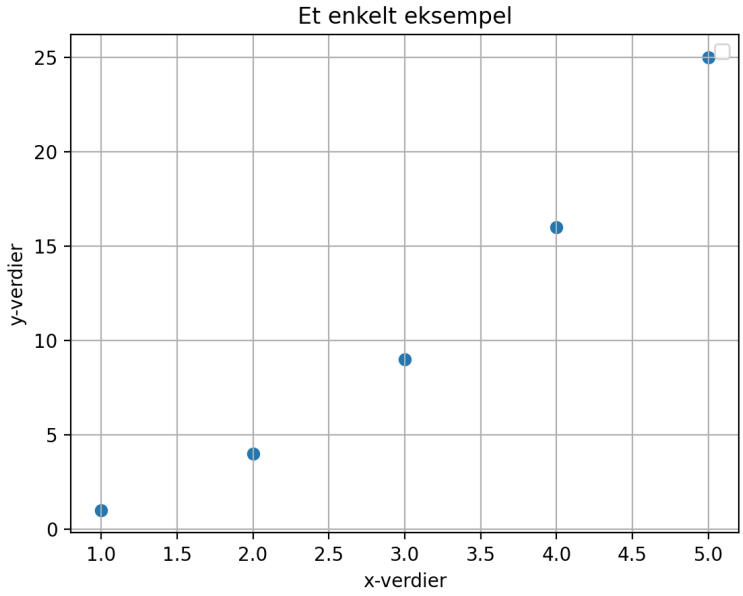
Resultatet vises i figur 9.2

Bare punktene

Nå har vi fått tegna opp punktene med linjer mellom punktene. Ofte er det akkurat det vi ønsker, men nå ønsker vi bare å få plottet punktene som et *punktplott*. På engelsk kalles det et *scatter plot*. Bytter vi ut `plt.plot(x,y)` med `plt.scatter(x,y)` sier vi fra om at det er et punktplott vi ønsker oss. Kjører vi programmet med den nye kommandoen får vi resultatet i figur 9.3.



Figur 9.2: Et enkelt plott med forklarende tekst



Figur 9.3: Et punktplott med forklarende tekst

Flere plott i samme koordinatsystem

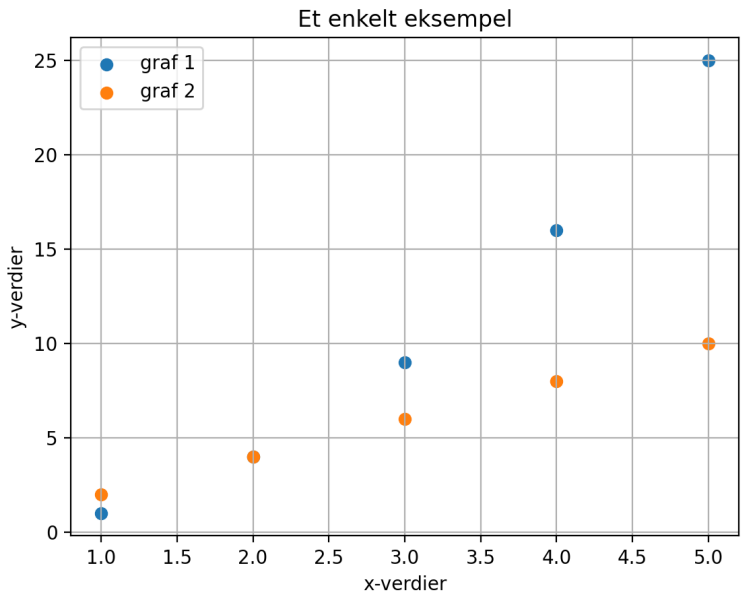
Punktplott kan inneholde flere serier med verdier som vi ønsker å plotte i samme koordinatsystem. Vi kan se på et eksempel hvor x-verdiene er de samme, men y-verdiene er ulike. I programkode 9.3 er det tre lister som inneholder verdiene. Verdiene plottes med å si fra om x- og y-verdier. I tillegg kan vi gi plottene navn. Kommandoen `plt.scatter(x,y1, label = "graf␣1")` sier fra om at det skal lages et plott og at plottet skal ha merkelappen (eng. label) «graf 1». For at vi skal få se navnene til de to seriene må vi si fra om det med `plt.legend()`.

```
1 import matplotlib.pyplot as plt
2 # legger inn x-verdier og y-verdier
3 x = [1,2,3,4,5]
```

```
4  y1 = [1,4,9,16,25]
5  y2 = [2,4,6,8,10]
6
7  # plotter verdiene
8  plt.scatter(x,y1, label = "graf 1")
9  plt.scatter(x,y2, label = "graf 2")
10
11 # tekst
12 plt.xlabel("x-verdier")
13 plt.ylabel("y-verdier")
14 plt.title("Et enkelt eksempel")
15 plt.legend()
16 plt.grid()
17 # viser plottet
18 plt.show()
```

Programkode 9.3: To ulike serier

Da får vi plottet to serier med data. Python velger ulike farger på de to plottene og vi får opp en forklaring med tekstene vi har valgt. Figur 9.4 viser resultatet.



Figur 9.4: Plott med to dataserier

9.2 Funksjoner som grafer

I matematikkundervisning får vi behov for å tegne grafer til matematiske funksjoner. Grafen til en funksjon består av en mengde x-verdier og de tilhørende funksjonsverdiene. For at det skal se ut som en graf og ikke et punktplott er vi avhengig av å ha mange x-verdier. De kan vi skrive inn, men det blir for mye jobb. Heldigvis kan vi få hjelp, men hjelpa må vi importere. Modulen NumPy inneholder en kommando som heter `linspace` som vi kan benytte til å lage mange x-verdier. Ved å si fra om en start- og en sluttverdi kan vi be om å få laget et visst antall verdier mellom dem:

```
linspace(start, stopp, antall)
```

Prøver vi kommandoen i terminalen kan vi få dette resultatet:

```
>>> import numpy as np
>>> x = np.linspace(-5,5,10)
>>> print(x)
[-5.  -3.88888889 -2.77777778 -1.66666667 -0.55555556  0.55555556 1.66666667
     2.77777778  3.88888889 5. ]
```

Vi får laget ti verdier som starter med -5 og slutter med 5. Kommandoen sørger for at verdiene er fordelt likt i intervallet. Resultatet kan vi betrakte som ei liste, men det er egentlig en egen datatype som heter `ndarray`.

La oss si at vi ønsker å tegne grafen til funksjonen gitt ved $f(x) = x^2$ i intervallet $[-10, 10]$. Da definerer vi først funksjonen i Python. Så må vi få laget oss x-verdier. Vi får tusen verdier i det gitte intervallet med `x = np.linspace(-10,10, 1000)`. Alle verdien tilordnes variabelen `x`. Innholdet kan vi betrakte som ei liste. Når disse verdiene sendes til den definerte funksjonen får vi tilbake alle de tilhørende y-verdiene. Nå er de klare for å plottes.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  # definerer funksjonen f
5  def f(x):
6      return x**2
7
8  # legger inn x-verdier
9  x = np.linspace(-10,10, 1000)
10
11 # beregner y-verdier
12 y = f(x)
13
14 # plotter
15 plt.plot(x,y)
16 plt.show()
```

Programkode 9.4: Plotting av graf

Grafen blir plottet i et koordinatsystem hvor aksene opprettes automatisk og vil se ut som i figur 9.6. Koordinatsystemet er kanskje litt uvant? Her er går ikke andreaksen gjennom origo, men tegnes til venstre. Det går an å gjøre om på det, men det krever en del kode. Foreløpig får vi nøye oss med denne framstillinga.

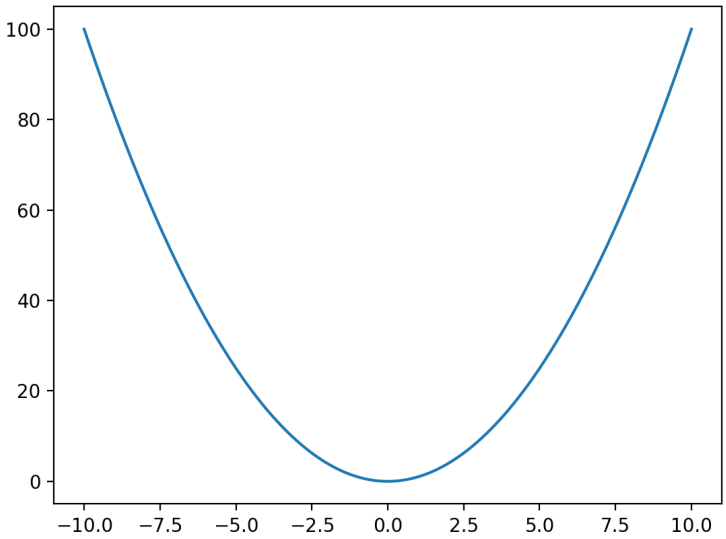
9.2.1 Flere grafer

Vi kan se på et eksempel hvor vi ønsker å tegne flere grafer i samme koordinatsystem.

Eksempel 1

Tegn grafene til disse funksjonene i intervallet $[-10, 10]$

$$f(x) = x^2$$
$$g(x) = 2x + 9$$



Figur 9.5: Grafen til $f(x) = x^2$

Framgangsmåten er som tidligere. Vi må definere funksjonene og vi må få laget x-verdier, grafene plottes ved å si fra om hvilke verdier vi vil ha med. Programkode 9.5 viser hvordan det kan bli gjort. Her er det også tatt med litt mer i plottet. Grafene har fått merkelapper og gitte farger. Måten vi kan skrive de matematiske symbolene krever en spesiell kode og at versjonen av Python vi benytter er i stand til å gjenkjenne det. Kommandoen er:

```
plt.plot(x,y1, label = r"$f(x)=x^2$", c = "r")
```

Her er `$f(x)=x^2$` koden for $f(x) = x^2$. Python velger forskjellige farger for ulike grafer, men vi kan bestemme hvilken farge vi ønsker med kommandoen `c = "<farge>"`. Vi skriver inn en bokstav for fargen. Det er bare å velge fra tabellen med bokstaver for de forskjellige fargene.

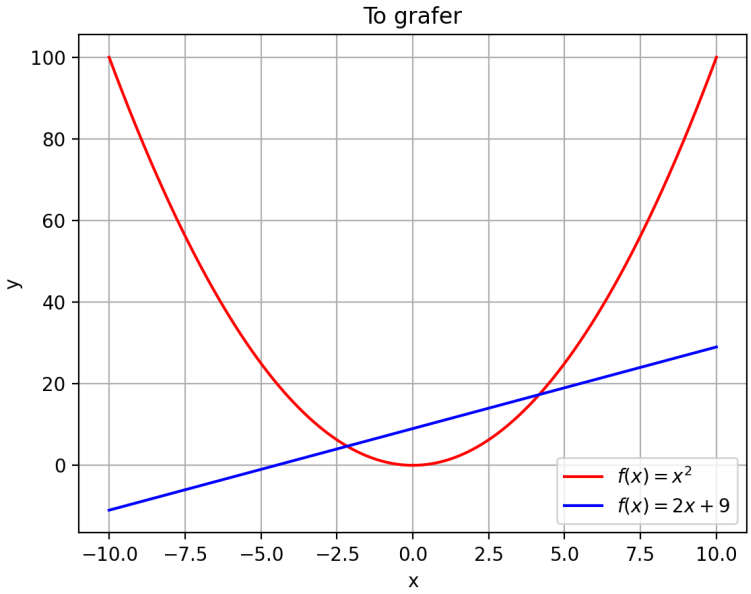
Farger

Forkorting	Farge
b	Blå
r	Rød
g	Grønn
c	Cyan
m	Magenta
y	Gul
k	Svart
w	Hvit

Tabell 9.1: Farger

```python
import matplotlib.pyplot as plt
import numpy as np

# definerer funksjonen f
def f(x):
    return x**2

def g(x):
    return 2*x+9

# legger inn x-verdier
x = np.linspace(-10,10, 1000)

# beregner y-verdier
y1 = f(x)
y2 = g(x)

# plotter
plt.plot(x,y1, label = r"$f(x)=x^2$", c = "r")
plt.plot(x,y2, label = r"$f(x)=2x+9$", c = "b")
plt.xlabel("x")
plt.ylabel("y")
plt.title("To grafer")
plt.legend()
plt.grid()
plt.show()
```

Programkode 9.5: To grafer med forskjellig farge



Figur 9.6: Plott med to dataserier

A Vedlegg

Vedleggene er litt forskjellig som kanskje noen finner interessant, men som ikke er av stor betydning for enkel programmering.

A.1 Formattering med f-strenger

Med versjon 3.6 introduserte Python noe som kalles f-strenger. Navnet kommer av at vi må skrive en f i kommandoen på denne måten `print(f"strengen")`. Når det er gjort kan vi legge til formatteringer av strengen. Har vi en variabel med navnet «x», som inneholder verdien til et nullpunkt, kan vi skrive ut den inne i strengen: `print(f"Nullpunktet␣er␣x␣=␣{x}")`

Det samme resultatet har vi oppnådd med bare `print` også, men det fine med f-strenger er at vi kan legge til mer formattering. La oss si at vi vil ha fire desimaler. Da kan vi skrive

```
print(f"Nullpunktet␣er␣x␣=␣{x:0.4f}")
```

Med f-strenger kan vi få utskriftene til å se penere ut. Det krever at vi benytter en versjon av Python etter 3.6. Har du ikke det, så går du bare glipp av litt pynt.

Vi kan se på et eksempel til. Programkode A.1 skriver ut en tabell med verdier.

```
1 def f(x):
2     return 2**x+3**(-x)-3
3
4 X = [-2, -1, 0, 1, 2]
5
6 print("01234567890123456789")
7
8 for x in X:
9     print(f"{x:7.0f} {f(x):8.2f}")
```

Programkode A.1: Funkjsonstabell

Programmet skriver ut en funksjonstabell.

```
01234567890123456789
   -2 6.25
   -1 0.50
    0 -1.00
    1 -0.67
    2 1.11
```

Kommandoen `print(f"{x:7.0f}␣{f(x):8.2f}")` formatterer strengen slik at desimalskilletegnet til `x` plasseres sju plasser til venstre. Her ber vi om at verdien skal skrives ut uten desimaler, og dermed desimalskilletegnet. Verdien som skrives ut skal være av type flyttall. Neste verdi plasseres med desimalskilletegnet åtte plasser til venstre for det som ble skrevet ut. Den verdien skal skrives ut med to desimaler.

Formatteringen med f-streng

```
f"{verdi:{bredde}.{presisjon}}"
```

Her er

- *verdi* er en variabel eller uttrykk som gir et tall
- *bredde* er bestemt av antall totalt antall tegn som settes av til det som skal vises
- *presisjon* er antall tegn som settes av etter desimalskilletegnet

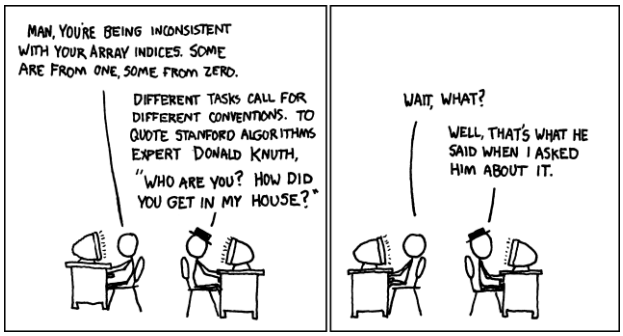
Dette gir oss mulighet til å formattere verdiene som skal skrives ut. Tabell A.1 viser en del eksempler

Verdi	Format	Utskrift	Forklaring
3.1415926	`{:.2f}`	3.14	Flyttall med to desimaler
3.1415926	`{:+.2f}`	+3.14	Flyttall med to desimaler inkludert fortegn
-1	`{:+.2f}`	-1.00	Flyttall med to desimaler inkludert fortegn
2.71828	`{:.0f}`	3	Flyttall uten desimaler
0.25	`{:.2%}`	25.00%	Formatter som prosent
1000000000	`{:.2e}`	1.00e+09	Skriv på standardform
13	`{:10d}`	xxxxxxxx13	Høyre-justert ned ti
13	`{:<10d}`	13xxxxxxxx	Venstre-justert
13	`{:^10d}`	xxxx13xxxx	Justert til sentrum og ti som bredde

Tabell A.1: Oversikt over noen formatteringer

Du kan finne mye mer på nettet om dette temaet. Litt eksperimentering hjelper også.

A.2 Nullindeksering



Noe som kan være forvirrende er at det er vanlig i programmering å starte indeksering med null. Det kalles *nullindeksering*. En slik nullindeksering kjenner mange også fra andre fag, men i hverdagen er vi vant med å starte telling med 1. Det kan være greit å skille mellom telling og indeksering. Et eksempel på nullindeksering kan være måten etasjer nummereres på engelsk. Der er vår første etasje *ground floor* og så starter de tellingen av etasjer over.

Edsger W. Dijkstra er en kjent nederlandsk informatiker. Han argumenterte for en nullindeksering i et notat fra 1982: Why numbering should start at zero. Der tok han for seg mulige måter indeksering kunne skje og viste til at nullindeksering har sine fordeler. Slik har det blitt vanlig å starte med null i programmeringsverdenen. Les gjerne mer her: Wikipedia: Zero-based numbering

A.3 Mer om funksjoner og parametre

Når vi benytter programmeringsfunksjoner overfører vi ofte parametre til funksjonen. Da har vi bestemt hvilke parametre vi ønsker å overføre og rekkefølgen på dem. Kanskje ønsker vi å endre rekkefølgen ved å spesifisere variabelnavnene når vi kaller funksjonen. Det kan vi få til ved å si fra når vi kaller funksjonen. Programkode A.2 viser hvordan kan spesifisere parametrene i den rekkefølgen vi ønsker.

```
1 def min_funksjon(fag3, fag2, fag1):
2     print("Jeg tar faget " + fag2)
3
4 min_funksjon(fag1 = "MGLU234", fag2 = "Lær2003", fag3 = "MGLU7654")
```

Programkode A.2: Navngiving av parametre

Noen ganger kan det være vanskelig å vite antall parametre, men Python gir oss noen muligheter i disse tilfellene også. Programkode A.3 viser et eksempel på hvordan vi kan definere en funksjon og si fra om at vi ikke vet hvor mange parametre som overføres. Ved at vi skriver `*fag` vil funksjonen motta parameterverdiene i ei liste og vi kan angi hvilket listeelement vi ønsker å benytte.

```
1 def min_funksjon(*fag):
2     print("Jeg tar faget " + fag[1])
3
4 min_funksjon("MGLU234", "Lær2003", "MGLU7654")
```

Programkode A.3: Paramteroverføring

Det er også en annen måte å gjøre det på ved at parametrene overføres i det som kalles en *dictionary*. Det gjør det mulig å finne fram til parameteren ved å skrive navnet, slik rrogramkode A.4 viser

```
1 def min_funksjon(**fag):
2     print("Jeg tar faget " + fag["fag2"])
3
4 min_funksjon(fag1 = "MGLU234", fag2 = "Lær2003", fag3 = "MGLU7654")
```

Programkode A.4: **kwargs

Ved paramteroverføring har vi også mulighet til å si fra om en standardverdi (*default value*) hvis det ikke overføres noen parametre. Overfører vi en parameter vil den verdien benyttes og ikke standardverdien. Programkode A.5 viser et eksempel.

```
1 def min_funksjon(fag = "Lær2003"):
2     print("Jeg tar faget " + fag)
3
4 min_funksjon()
5 min_funksjon(fag = "MGLU234")
6 min_funksjon(fag = "MGLU7654")
```

Programkode A.5: Standardverdi

Du kan finne ut mye mer om dette ved å søke på nettet.

Figurer

Tabeller

Pythonkode