Kurs i programmering for grunnskolelærere

En kort innføring i programmering og Python

kodeskolen simula

kodeskolen@simula.no

Dette kompendiet er kursmateriale for et kurs i programmering for lærere som har liten, eller ingen, tidligere erfaring med programmering.

I løpet av kurset vil vi gi en kort innføring i hvordan man kan bruke det tekstbaserte programmeringspråket Python som et verktøy i eget klasserom. Grunnet begrenset med tid vil vi ikke prøve å gi en heldekkende innføring i programmering, eller generelle bruksområder, det har vi rett og slett ikke tid til. Derimot prøver vi å gi en innføring som er tilstrekkelig til at dere kan begynne å leke dere litt med programmering og begynne å få idéer om hva slags leker, oppgaver og prosjekter man kan få til i klasserommet med programmeringen.

Eksemplene vi bruker prøver vi å legge på et nivå slik at de passer for 5.–10.-klasse, og i denne omgang legger vi spesielt fokus på noen utvalgte tema i matematikken.

Programmering læres best av at man prøver selv. Det er viktig å gjøre oppgaver, eksperimentere med stoffet og prøve seg frem. Når man programmerer kommer man til å gjøre mye feil. Dette gjelder både når man er helt fersk eller skikkelig erfaren. Det er derfor viktig at du tør å prøve deg frem, og at du spør om hjelp og råd når du kjører deg fast. Vi er her for å hjelpe, og det er alltid lurt å diskutere med kollegaer, selv om de kanskje ikke kan noe mer enn deg. Programmering er ikke en aktivitet som gjøres best alene, tvert imot. Vi er mest effektive når vi jobber sammen og diskutere problemstillinger, idéer og fremgangsmåter.

Innhold

1 Algoritmisk tankegang

1.1 Hvorfor programmering?

> Informatikk handler ikke mer om datamaskiner enn astronomi handler om teleskoper.
>
> *Edsger Dijkstra*

Datamaskiner og moderne teknologi har gitt oss uvurderlige verktøy – måter å utføre mindre og større oppgaver på. Noen verktøy løser disse på en mer elegant og effektiv måte enn vi kunne ha klart uten programmering. Andre løser problemer man før anså som umulige å løse – fordi det krevde såpass mange steg, så mange beregninger, eller lignende.

Når vi programmerer er fokuset vårt som oftest rettet mot et problem, ikke mot hvordan datamaskinen løser dette, selv om begge deler kan være interessant. Det blir litt på samme måte som med matematikk: Ofte løser vi et problem, men det kan også være interessant for sin egen del.

Programmering er i likhet med matematikk basert på logikk. Det gjør at man enkelt kan bruke matematikk som et verktøy når man programmerer, og at man kan løse matematiske ligninger ved hjelp av programmering. Veldig mye at det man kan få til på en datamaskin går igjennom et matematisk rammeverk. Hvis man skal jobbe med noe som finnes i naturen, f.eks. å forutsi været, å simulere hvordan hjertet fungerer, eller å animere havbølger i en film, så må det først defineres presist matematisk. Deretter kan problemet løses ved hjelp av en datamaskin. Uten matematikken ville ikke datamaskinen klare å løse disse problemene, men uten datamaskinen hadde vi ikke klart å løse alle ligningene som omfattes av slike problemer.

Når vi underviser i programmering tenker vi selvfølgelig heller ikke at alle elevene skal bli programmerere. Men vi lever i et samfunn der mange problemer løses ved hjelp av programmering, og da vil det være nyttig å forstå hva som skjer, og hvordan dette skjer.

1.2 Teknologiens ABC

A is for algorithm
B is for (boolean) logic
C is for creativity and computers

Linda Liukas

1.2.1 Algoritmer

Datamaskiner har ingen vilje og trenger instruksjoner for å virke. Alt en maskin gjør er bygd opp av en rekke logiske instrukser. Dette krever at vi som mennesker, som gir disse instruksjonene, er ytterst nøyaktige i våre beskrivelser. En algoritme er en slags oppskrift, en trinnvis beskrivelse, av hva som skal utføres.

For å forstå hvordan algoritmer er bygd opp kan man jobbe i par der den ene utfører og den andre gir instruksjoner. Den som utfører forteller den andre det som skal skje. Prøv for eksempel:

- Å gå fra et punkt i rommet til et annet.
- Å bevege seg på et rutenett (på et ark, eller tegn opp med kritt ute), med et utgangspunkt og et mål. Prøv først med at instruktør ser det som skjer, deretter uten.
- Å tegne, eller kopiere en tegning, rygg til rygg, der den ene forklarer, den andre tegner etter instruksjoner.
- Å bygge legofigurer, eller kopiere en legofigur, rygg til rygg, der den ene forklarer, den andre tegner etter instruksjoner.

Den som utfører har ikke lov til å stille spørsmål. Dere vil helt sikkert komme i situasjoner der resultatet ikke var som forventet. For eksempel glemte man en legokloss i beskrivelsen og da må man gå tilbake og finne ut hva som var feil i instruksjonen. Dette kalles *feilsøking* og utgjør en stor del av programmering. Når det etterhvert fungerer skal man ha funnet en oppskrift som er nøyaktig nok til at hvem som helst skal kunne følge den.

Algoritmer er «overalt» selv om man kanskje ikke tenker over at de er her. Hvis du søker på Google, bruker Google algoritmer for å finne relevante søkeresultater

– og de bruker algoritmer for å finne de fortest mulig. Hvis du skal kjøre et sted, bruker GPS-en algoritmer for å finne korteste kjørevei til målet. Hvis du har sett filmer på Netflix eller hørt på musikk på Spotify, bruker disse algoritmer for å finne lignende filmer eller musikk som de foreslår at du også vil kunne like.

1.2.2 Boolsk logikk

Datamaskiner er logiske. Alt de gjør følger mønstre bestemt av om noe er *sant* eller *usant*. Dette er laget i flere nivåer av *abstraksjon* slik at man kan forholde seg til en overkommelig mengde av informasjon til enhver tid.

Boolsk logikk er basert på noen grunnkonsepter. En tilstand kan være *sann* eller *usann*. Man kan kombinere denne tilstanden, kall den A, med en annen tilstand, si B. Derfra kan man få en tredje tilstand ved å kombinere A og B. Her kan

- $C = A$ **og** B. Da er C sann hvis både A og B er sanne.
- $C = A$ **eller** B. Da er C sann hvis enten A og/eller B er sanne.

Man kan også få en tilstand ved en *negasjon* – en *ikke* A. Så hvis $D =$ **ikke** A, er D sann hvis A er usann, og motsatt.

Dette er faktisk alt som trengs for å bygge opp en datamaskin. Det starter med at elektriske signaler blir utvekslet eller ikke. Kombinerer vi lag på lag av dette får vi en maskin med utallige muligheter og bruksområder.

1.2.3 Kreativitet og datamaskiner

Når man tenker på kreativitet tenker man kanskje på kunst som malerier, poetiske tekster og skuespill. Når man tenker på datamaskiner tenker man kanskje på verktøy som tekster, tall og databaser. Men alt en datamaskin er, kommer av at noen var kreative. Datamaskiner er resultatet av at noen kreative mennesker forsto hvordan de skulle kombinere boolsk logikk til noe håndterlig.

Når vi programmererer er det viktig å *finne* måter å løse problemet på. Det hjelper ofte å tegne, å ta bort teksten og tallene, og isteden bruke piler og bokser. Man må være kreativ for å *formulere* problemet: «Hva er det som er interessant her?». Man må være kreativ for å *løse* problemet: «Hvilke ulike muligheter har vi, og

hvilke steg kan vi kombinere på hvilken måte?». Og man må være kreativ i måten man *presenterer* det på: «Hva vil vi at datamaskinen skal vise oss til slutt?».

1.3 Hvordan programmere?

> Dataprogrammereren er skaperen av universer hvor han alene bestemmer reglene. Ingen dramatiker, ingen sceneregissør, ingen keiser, har noen gang hatt slik absolutt autoritet til å arrangere en scene eller kampfelt og til å kommandere slik stødige og trofaste skuespillere eller tropper.
>
> *Joseph Weizenbaum*

1.3.1 Designe et program

Programmering kan ofte deles opp i tre deler:

1. Finne ut hva problemet er.
2. Finne ut hvordan man kan løse dette problemet.
3. Implementere løsningen.

Først må du altså vite hva utfordringen er. Deretter, og dette er ganske viktig, finne ut hvordan du skal løse det. Til slutt kommer selve løsningen. Den kan vise seg å være relativt enkel dersom du gjorde det du skulle på punkt to. Dette kan sammenlignes med hvordan man kan løse en matematikkoppgave, der man kan dele opp stegene på samme måte:

1. Hvor mange epler kan du kjøpe for 200 kr, hvis de koster 4,50 kr pr.?
2. Hvordan kan du dele 200 på 4,50?

Figur 1: Den algoritmiske tenkeren. Figuren er laget av Udir, som igjen har tilpasset den fra Barefoot Computing (UK), publisert med en åpen lisens (OGL).

3. Utfør beregningen 200 / 4,50.

Steg to kan gjerne gjennomføres på papir, og ikke på en datamaskin. Man kan tegne, skissere og komme med ideer, uten å sette seg fast på syntaks og hvordan noe skrives i et bestemt språk. I større programmer bryter man gjerne ned programmet til flere underproblemer og bruker fremgangsmåten over flere ganger og på forskjellige nivåer.

1.3.2 Hvordan løse et problem?

I fremgangsmåten for steg to beskrevet over er det en del konsepter som går igjen på tvers av de ulike problemene man prøver å løse.

1. Man må tenke *logisk*: Man må kunne se sammenhengene mellom formuleringen av problemet, hvordan problemet kan løses, og selvfølgelig klare å følge denne logikken i implementeringen.
2. Man må tenke *algoritmisk*: Man må klare å gå fra et noe mer eller mindre konkret beskrevet problem til en steg-til-steg-oppskrift.

3. Man må kunne *dekomposisjon*, man må bryte ned problemet til flere mindre oppgaver.

4. Man må finne *mønstre*: Finne ut hvilke deler av problemet som kan løses på samme måte.

5. Man må kunne *abstraksjon*: Kunne se sammenhenger utover det konkrete problemet – ikke «Hva er 200 delt på 4,50?», men «Hvordan deler man et tall på et annet?».

6. Man må kunne *evaluere* løsningen sin underveis og til slutt vurdere om dette er et fornuftig steg og om det gir riktige resultater.

1.3.3 Arbeidsmåter

Når man går over til implementasjon er det igjen flere ting vi har til felles på tvers av hva vi jobber med. For å løse problemene har programmerere visse måter å jobbe på:

1. Det er mange små detaljer som skal på plass, fordi man må være helt nøyaktig og presis i sin kommunikasjon. Evnen til å *fikle* er viktig.

2. Man må ha som perspektiv at man skal *skape*, designe og lage noe for å løse noe. I større grad enn vanlig i matematikk må man være kreativ og finne på noe nytt.

3. ... men på samme måte som når man har en feil i et matematikkstykke, må man gå tilbake og finne ut hva som er feil hvis man får resultater som ikke stemmer. Det er lett for et menneske å gjøre små logiske feil, og det er helt vanlig, men for å få programmet til å fungere må man finne feil gjennom *feilsøking*.

4. Programmering kan være en tålmodighetstest uten like, og det som kjennetegner de som kommer til å jobbe videre med det er ofte evnen til å *holde ut* – en stahet, en overbevisning om at *det skal jeg da klare*!

5. Men selv om man gjerne vil klare det, er det usedvanlig viktig å *samarbeide* i programmering. Det har to aspekter: Man kan få hjelp til å tenke på andre måter ved å bruke den andres kompetanse, men også er det til stor hjelp for deg selv å faktisk forklare et problem til en annen. Man bruker en annen del av hjernen enn den man ville brukt dersom man forsøkte å løse et problem på egen hånd.

1.4 Grunnleggende konsepter

1.4.1 Input og output

To viktige konsepter for algoritmer er som på engelsk kalles for *input* og *output*. Disse oversettes noen ganger med «inndata» og «utdata», men de engelske låneordene har også blitt godt innarbeidet i norsk språk. *Input* er det man gir inn som informasjon til algoritmen og man skal ta utgangspunkt i. For eksempel vil input til et Google-søk være ordene man søker etter. Input til GPS-en vil være hvor du vil starte, og hvor du vil avslutte reisen. *Output* for søkeresultatet vil være listen med nettsider, og for navigasjonen en beskrivelse av ruten. Hva er *input* og *output* for algoritmen som finner lignende filmer eller musikk?

1.4.2 Variabler

Variabler er datamaskinens måte å «huske» ting på. Det er knagger der man tar vare på utregninger man allerede har utført. Det er også måten vi kan kommunisere med datamaskinen på. Vi kan be datamaskinen huske på en verdi som vi gir et navn, og vi kan be datamaskinen gi tilbake en annen verdi i from av en variabel.

Ofte er det lurt å huske på å gi beskrivende, men korte navn til variabler. Da er det lett for oss mennesker å huske på hva som er hva og forstå logikken hvis vi trenger å gå igjennom programmet igjen på et senere tidspunkt.

1.4.3 Tester

En datamaskin trenger å vite hva den skal gjøre og det må være presist definert. Mellom mennesker er det ofte underforstått hva som skal skje. Hvis man ber et menneske vaske opp kan vedkommende selv finne ut hvor mye vann som skal til for å vaske noe, hvor lenge det skal vaskes og selv vurdere når noe er rent nok til å settes over til oppvaskstativet. Hvis du skal programmere en vaskemaskin må maskinen ha beskjed om hvor mye vann du skal bruke og over hvor lang tid. Hvis maskinen er litt avansert har du kanskje et program som selv finner ut om maskinen er ren nok eller om det trengs mer tid og vann. Altså *hvis* vannet man sender ut virker skitten, så trenger vi en sensor og en definisjon av hva som er skittent og hva som er rent, om programmet skal fortsette. Kanskje har vi en timer som sier at vi ikke skal vaske i det uendelige uansett, så *hvis* vannet virker skittent og *hvis*

tiden ikke har blitt mer enn 2 timer, fortsetter vi programmet.

Når vi tester noe anvender vi boolsk logikk direkte. Vi tester om en betingelse er *sann* eller *usann*. Det kan for eksempel være om et tall er mindre enn et annet, eller om en som bruker programmet ditt har gitt en spesiell verdi som input. Vi kan lett kombinere flere uttrykk ved *og*, *eller* og *negasjon*.

Dette er det lett å finne mange eksempler på fra dagliglivet. *Hvis* klokken er syv, da ringer vekkerklokka. *Hvis* det er grønt lys, så kan du gå eller kjøre. *Hvis* man trykker på både denne *og* denne knappen på en maskin med barnesikring, så går barnesikringen av.

Frakoblet: Rødt lys, grønt lys

Dette er en enkel lek man kan gjøre for å forsterke begrepene rundt hvis-setninger og betingelser.

Leken går ut på at elevene stiller seg på en ene enden av klasserommet, eller ute, og skal komme seg over til den andre enden. Man kan alternativt spille det på ruteark, og rett og slett krysse av for hvert skritt man går.

Måten leken utføres er at læreren, eller en elev, leser opp betingelser. For eksempel:

- **Hvis** du har på deg olabukse, **så** ta to skritt frem
- **Hvis** du har bokstaven «e» i navnet ditt, **så** ta et skritt frem, **ellers** ta et skritt tilbake
- **Hvis** du har blå øyne **eller** brune øyne, **så** ta et ekstra langt skritt frem

Her kan man skille mellom rene hvis-så setninger, eller hvis-ellers setninger. Man kan også inkludere konseptene OG og ELLER.

Når første person kommer i mål kan vinneren kåres. Her kan man for eksempel la vinneren lage betingelsene for neste runde. Pass på at betingelsene som lages er sann for hvertfall et par elever.

Etter man har lekt leken et par ganger bør man reflektere tilbake på begrepsbruken. Her kan man gjerne knytte opp de norske begrepene man har brukt med de engelske begrepene.

1.4.4 Løkker

En av datamaskinens største fordeler er evnen til å repetere noe om og om igjen. Ber du en datamaskin legge sammen 1000 tall skjer det raskere enn du rekker å skrive inn kommandoen for å gjøre det. Hvor lang tid hadde det tatt for et menneske? Dette er en oppgave som er lett for en maskin fordi den er presist definert og man utfører *akkurat samme* operasjon et bestemt antall ganger. Du starter med en sum på 0, og *mens* det fortsatt er tall igjen, legger du til disse i summen, en etter en. Hver slik operasjon kalles for en *iterasjon*.

Løkker henger sammen med tester i den forstand at i en løkke er det alltid en test – enten direkte definert eller implisitt. Man kan tenke på det som at *hvis* en betingelse er sann fortsetter vi å gjøre noe. Hvis betingelsen er usann stopper vi.

Vi skal gå igjennom to typer løkker, som egentlig kan sees på som varianter av hverandre. Den ene typen løkker foregår *mens* noe er sant, med en test før hver iterasjon. *Mens* vannet ikke er rent nok, fortsetter vi vaskeprogrammet. Den andre typen løkker gjentar noe et visst antall ganger. Her kan man kanskje tenke på et treningsprogram. Man kan for eksempel sette opp at man skal løpe ti intervaller (*gjenta* et intervall *ti ganger*), eller gjøre en gitt styrkeøvelse femten ganger.

Frakoblet aktivitet: Navigasjon

Dette er en enkel øvelse du kan gjøre i klasseromme for å introdusere konseptet om bruk av repetisjon og løkker i algoritmer.

Sett en stol eller et annet objekt på gulvet med litt god plass rundt seg. Stell deg så litt bak og til siden for objektet.

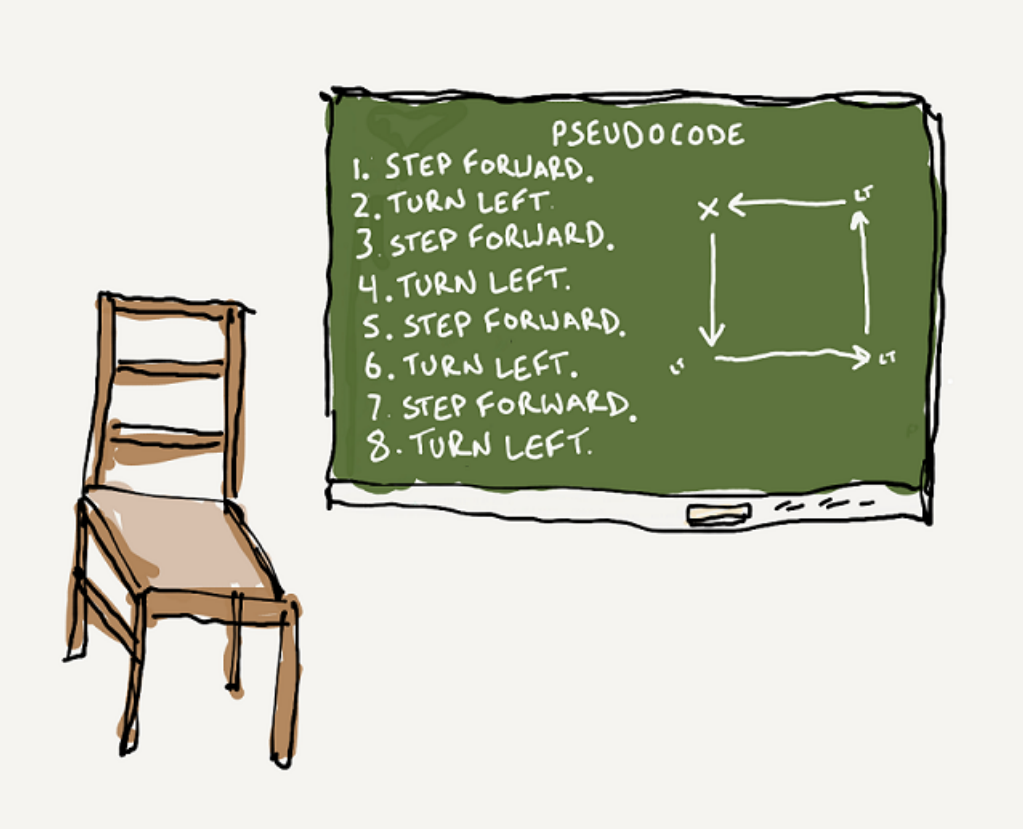
Fortell elevene at de nå må lage en *algoritme* eller et *program* til deg, for å navigere rundt stolen, og ende opp der du startet.

La elevene gjette på mulige instrukser. Be dem gjerne være mer spesifikke med instruksene sine. Isteden for å si 'Gå fremover', kan de for eksempel 'Ta ett skritt fremover'. Istedenfor å si snu til venstre, kan de si 'Snu 90 grader til venstre'. Skriv opp instruksene som ser ut til å løse oppgaven opp på tavlen så vi får et program i *pseudokode*. Altså kode som ikke er skrevet i et gitt programmeringsspråk, men med vanlig norsk eller engelsk.

Programmet for å navigere rundt stolen blir til slutt gjerne 8 steg:

```
Gå to steg frem
```

```
Snu 90 grader til venstre
Gå to steg frem
Snu 90 grader til venstre
Gå to steg frem
Snu 90 grader til venstre
Gå to steg frem
Snu 90 grader til venstre
```



Figur 2: Å navigere rundt en stol er en enkel oppgave, men hvis man ikke bruker repetisjoner blir det fort mange instrukser! Bildet er tatt fra makecode.microbit.org/courses/csintro/iteration/unplugged

Snakk nå med elevene og spør om vi kan gjøre dette programmet noe enklere eller kortere. Målet er å komme frem til at vi gjennomfører de samme to stegene, gang på gang. Derfor kan vi bytte ut programmet vårt med

```
Gjenta 4 ganger:
    Gå to steg frem
    Snu 90 grader til venstre
```

Her er nøkkelordet *gjenta*. Ettersom at koden vår nå gjentar seg flere ganger har vi laget en *løkke*.

Denne øvelsen trenger ikke å ta så lang tid, men kan hjelpe elevene med å skjønne konseptet om gjentagelser, og at de kan spare oss tid og gjøre koden vår mer effektiv.

2 Vi setter igang med Python

Vi skal nå begynne å se på programmeringsspråket Python. Dette er et tekstbasert programmeringsspråk, som betyr at vi skriver koden ut som tekst, som datamaskinen så tolker når programmet kjøres.

Akkurat som vanlig språk finnes det mange ulike programmeringsspråk man kan lære seg. Hovedgrunnen til at vi velger å fokusere på Python, er at vi mener dette er et tekstbasert språk som passer spesielt godt om man skal fokusere på tekstbasert programmering i skolen. Dette er fordi det anses som mer nybegynnervennlig, og det er plattformuavhengig. Man kan derfor kjapt komme igang med enkle programmer og eksempler. Selv om man bare ønsker å bruke enklere former for programmering, som blokkbasert, i eget klasserom, anbefaler vi lærere å sette seg inn i grunnleggende tekstbasert programmering i Python.

I dette kapittelet vil vi først gå igjennom det du trenger av programvare for å jobbe med Python på din egen maskin. Her finnes det ulike valg avhengig av hvilken plattform du jobber på, og personlig preferanse. Deretter tar vi deg steg for steg igjennom ditt første Pythonprogram, hvor vi forklarer steg for steg hvordan du skriver kode, lagrer programmet, kjører det og ser resultatet.

Dette kapitellet er hovedsakelig ment for de som aldri har jobbet med Python eller tekstbasert programmering før. Vi har inkludert det for å gjøre det lettere å komme igang. I de følgende kapitlene vil vi begynne å dekke Pythonprogrammering mer detaljert.

2.1 Nødvendig programvare

For å programmere i Python vil du trenge to ting på maskinen din: et redigeringsprogram, hvor du kan skrive selve koden din, og et tolkeprogram, som kan tolke Pythonkoden og oversette dette til maskininstrukser som datamaskinen så kan gjennomføre. Du må altså kunne både *skrive* og *kjøre* Pythonkode.

Ettersom at Python utvikles som et fri programvareprosjekt, kan hvem som helst lage sine egne programmer for å jobbe språket, og derfor finnes det også en rekke ulike muligheter man kan velge blant. Akkurat hva du bør installere og bruke av programvare avhenger av hva slags plattform du jobber på, hva slags programmering du skal drive med, og personlig preferanse.

Uavhengig av hva slags plattform du jobber på anbefaler vi å gå for en løsning der du redigerer og kjører koden din i samme program. Dette gjør det enklere for nybegynnere å jobbe med kode. I de neste avsnittene dekker vi de programvarene vi anbefaler for de vanligste plattformene.

I resten av dette kompendiet vil vi vise mange eksempler på Pythonkode og resultatene de gir. Derimot vil disse eksemplene vises i en generell form og vil ikke være knyttet opp mot spesifikk programvare.

2.1.1 Ulike versjoner av Python

Uten å gå for mye inn på detaljer bør det nevnes at det finnes ulike versjoner av Python, og du vil muligens måtte velge versjon når du skal installere programvare. De to hovedversjonene av Python som ser bruk i dag er Python 2 og 3. Vi anbefaler på det sterkeste å kun bruke Python 3. Hovedgrunnen til å bruke den eldre versjonen, Python 2, er av kompabilitetsgrunner. Uten å gå inn på alle forskjeller mellom de to, kan vi nevne at Python 3 blant annet støtter unicode, som betyr at vi fritt kan bruke norske bokstaver (altså 'æøå') og andre spesielle karakterer i koden. I Python 2 er dette mye strengere og man blir i utgangspunktet tvunget til å programmere med kun engelsk alfabet.

2.1.2 Python på datamaskin: Anaconda

Om du skal programmere på en datamaskin, uavhengig om det er i Windows, Mac eller Linux, har du et par ulike muligheter. En enkel løsning er å laste ned program-

pakka *Anaconda Distribution*[1]. Dette er en gratis samlepakke som vil installere Python, redigeringsprogrammet Spyder, og andre tilleggspakker og småprogrammer som kan være nyttige, spesielt om man jobber med realfag.

Første gang du åpner Spyder ser det hele gjerne litt komplisert ut, du kan se en skjermfangst i Figur 3. Kort fortalt er Spyder pakket full med funksjonalitet som er praktisk for erfarne programmerere, men som vi skal mer eller mindre ignorere. Det som er viktig å merke seg på dette tidspunktet er at programmet består av forskjellige vinduer. Det større vinduet på venstre side av skjermen (1) er redigeringsvinduet. Det er her vi skriver selve koden vår, det kalles gjerne for en *Editor* på engelsk. Det mindre vinduet nede til venstre (2) kalles for en *konsoll*, og det er her outputten eller resultatene fra programmene våres vil vises. Vinduet øverst til høyre (3) er et hjelpevindu som kan gi tilleggsinformasjon.

Informasjonen i hjelpevinduet kan være mer til distraksjon en faktisk hjelp for nybegynnere, og du kan derfor godt lukket dette vinduet fullstendig, ved å klikke et par ganger på korset øverst til høyre i hjelpevinduet.

2.1.3 Python på datamaskin: IDLE

Et alternativ til å laste ned Anaconda, er å laste ned Python direkte fra utviklersiden `Python.org`[2]. Her får du en helt grunnleggende Pythoninstallasjon, tileggspakker for å for eksempel drive med turtleprogrammering må isåfall installeres separat. I tilegg til Pythoninstallasjonen får du redigeringsprogrammet IDLE. Sammenlignet med Spyder er IDLE langt mer minimalistisk, men dette kan fungere godt for nybegynnere.

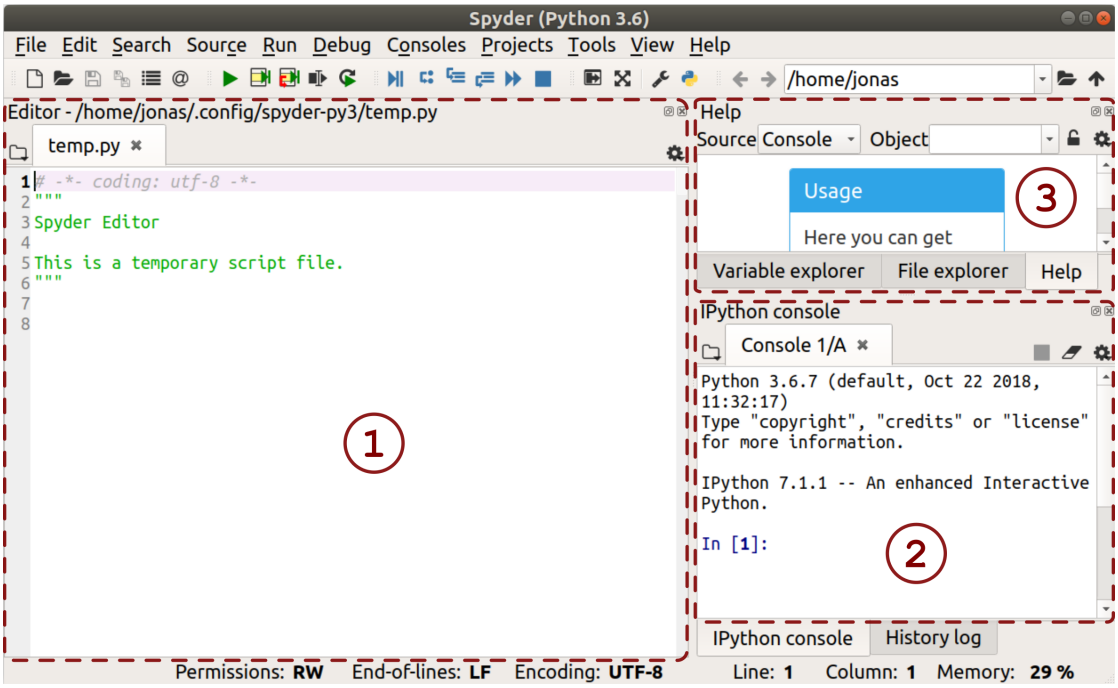
2.1.4 Python i nettleser: Trinket

Et alternativ til å installere programvare på egen maskin er å programmere i nettleseren sin. Det finnes flere ulike nettsider som lar deg både skrive og kjøre Pythonkode på nett. Dette er et alternativ om man ikke har mulighet til, eller ønsker, å installere programvare.

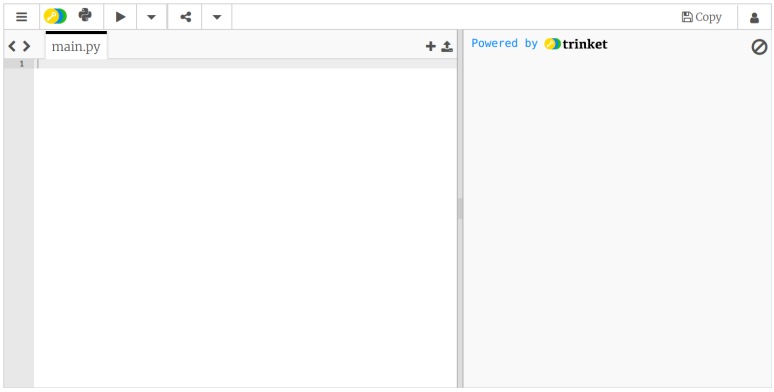
Et eksempel på et slik nettside er *Trinket*. Ved å gå inn på nettsiden

[1] Du kan laste ned denne pakken fra `https://www.anaconda.com/distribution/`

[2] Gå til `https://www.python.org/downloads/windows/` for Windows, og `https://www.python.org/downloads/mac-osx/` for Mac.



Figur 3: En skjermfangst av hvordan Spyder kan se ut ved første bruk. I tilegg til menyer og en verktøylinje på toppen, består programmet av ulike vinuder. Redigeringsvinduet (1) brukes til å skrive selve Pythonkoden vår, konsollen (2) er der vi får resultatene når vi kjører et program, og (3) er et hjelpevindu hvor vi kan få tileggsinformasjon.



Figur 4: En skjermfangst av nettsiden `trinket.io/python3`. Vinduet til venstre er der vi kan skrive koden. Vinduet til høyre viser resultatene av en kjøring.

- `https://trinket.io/python3`

Kommer du rett inn i en interaktiv sesjon hvor du kan skrive og kjøre Pythonkode. En skjermfangst vises i Figur 4.

Trinket kan brukes gratis, uten å lage bruker. Derimot får man utvidet funksjonalitet ved å lage en bruker, som for eksempel å lagre kode for senere bruk. Det finnes både gratisbrukere og betaltversjoner med mer fordeler. Det kan være veldig praktisk å lage en gratisbruker for lærer. I praksis kan det være utfordringer for elever å lage brukere grunn av personvernhensyn.

Det finnes lignende sider som Trinket som lar brukere programmere Python uten å lage bruker. Et annet eksempel er `Repl.it`.

2.1.5 Programmering på Chromebook

Om man ønsker å programmere med Python på Chromebook har man hovedsakelig to valg. Man kan enten gå for en nettbasert løsning som Trinket. Eller man kan bruke *Coding with Chrome*, en applikasjon for Chrome som er utviklet av Google. Coding with Chrome fungere med både Python og Javascript, men også det blokkbaserte språket blockly.

2.1.6 Python på nettbrett

For å programmere på nettbrett er nettbaserte løsninger fortsatt en god mulighet. Det finnes også visse applikasjoner som lar det skrive og kjøre Pythonkode, desverre er ikke alle disse av veldig god kvalitet, eller er like nybegynnervennlige. Om du er på iPad kan vi derimot anbefale applikasjonen Pythonista 3, som er godt utviklet, og har et intuitivt brukergrensesnitt. I motsetning til de andre løsningene vi nevner har Pythonista kun en lisensiert versjon som koster penger.

2.2 Ditt første Pythonprogram

Om du har klart å installere Python på ditt system, er du nå klar for å prøve å skrive og kjøre Pythonkode. Det er på tide å lage ditt første Pythonprogram.

Som ditt første program kan du lage et såkalt *"Hello, World!"*-program. Et slikt program er ikke fryktelig komplisert, det skal bare skrive ut en beskjed når det kjøres. Dette er et vanlig første program å skrive. Ikke bare for de som lærer seg å programmere, men det er også en vanlig øvelse når man skal lære seg et nytt programmeringsspråk, eller har gjort en fersk installasjon. Det er programmeringsverdens svar på å si "Hei, mitt navn er..." på et nytt språk.

For å begynne å skrive et slikt program må vi opprette en ny fil. Ethvert Pythonprogram vil være sin egen fil, og slike filer lagres med filendelsen ".py" på en datamaskin. Om du programmerer i nettleser kan vi ikke lagre filene på samme måte. Uavhengig av hvilket program du jobber i fil det finnes en form for "Ny fil"-knapp, mest sannsynlig i verktøylinjen på toppen av programmet. Det lønner å gi programmene du lager beskrivende navn, så det er lettere å holde oversikt når du har skrevet mange ulike programmer. Ettersom at vi laget et "Hello, World!"-program, kan du for eksempel velge navnet `hello.py`.

I den nye filen vår skal vi nå skrive koden vår. I Python er koden vi trenger for å skrive ut beskjeden "Hello, World!" ganske enkel, vi trenger faktisk kun én kodelinje:

```
1 print('Hei, Verden!')
```

Når du har skrevet koden din inn i editoren, og lagret programmet ditt som en fil. Kan vi *kjøre* koden. Det vil si å be datamaskinen gjennomføre programmet. I alle våre anbefalte programvarer vil dette gjøres ved å trykke en "Kjør"-knapp

du finner i verktøylinja eller lignende, den har mest sannsynlig en gjenkjennelig trekantikon.

Om du har skrevet koden riktig, og alt fungerer som det skal, så skal du nå få ut beskjeden i konsollen, som er vinduet til høyre.

```
Hei, Verden!
```

Om noe er galt med koden din vil du istedet få en *feilmelding*, som prøver å forklare hva som er galt. Vi vil diskutere disse feilmeldingene litt mer senere, men for nå anbefaler vi bare å dobbeltsjekke at du har skrevet nøyaktig det som står i eksempelkoden. Om du har skrevet nøyaktig det samme som står over. Om du har forsikret deg om at du har skrevet nøyaktig det samme som i eksempelkoden, og du fortsatt får en feil, kan det være at noe er installert feil. Spør isåfall gjerne om hjelp.

I `hello.py` eksempelprogrammet bruker vi *funksjonen* `print` til å skrive ut en beskjed. Med *print* mener vi her ikke å skrive ut til papir på en printer, men istedet å skrive ut informasjon til konsollen, slik at vi kan lese den på skjermen. Med denne funksjonen kan vi altså skrive beskjeder og gi informasjon til oss når vi kjører et program, og det er ofte slik vi vil dele resultatet av en utregning for eksempel.

For å bruke `print` , så må vi fortelle datamaskinen *hva* den skal printe, og dette gjør vi ved å legge selve det som skal printes i parenteser, `()` , bak selve `print`-kommandoen. Bruken av parenteser på denne måten går igjen hver gang vi bruker en funksjon, dette kommer vi tilbake til senere.

Inne i parentesene skal vi nå skrive beskjeden vår. Så der skriver vi `'Hei, Verden !'`. Her må vi være tydelige, så datamaskinen ikke prøver å tolke selve beskjeden som kode, for da blir den forvirret. Vi bruker derfor apostrofer, eller *fnutter*, som vi gjerne kaller dem (`'`) , rundt selve beskjeden. Dette kaller vi for en *tekststreng*. Beskjeden vi skriver er altså en tekst, og ikke en kode, derfor blir den også farget blått i eksempelet vårt, for å lettere skille den fra resten av koden vår. Vi kan også bruke anførselstegn (`"`) til akkurat samme formål.

Merk at fargene på koden du skriver ofte kan være ulik fra fargene slik de er i eksemplene her i kompendiet. Forskjellige programmer bruker gjerne ulike farger. Det er ikke viktig akkurat hvilke farger de forskjellige delen av koden er. Det som er viktig er at fargene kan gjøre det lettere å skille mellom de ulike delene av koden, og gjør det lettere å finne feil.

2.2.1 "Hello, World"-programmet i Java

Som nevnt er det vanlig å bruke "Hello, World!"-programmet som et slags "Hei, mitt navn er...". Det er altså et slags eksempel på hvordan er programmeringsspråk ser ut og den grunnleggende *syntaksen*. La oss derfor kort vise det samme programmet skrevet i programmeringsspråket Java, det kan for eksempel se ut som følger

```
1 public class Hello {
2     public static void main(String[] args) {
3         System.out.println("Hello, World!");
4     }
5 }
```

Om du sammenligner Pythoneksempelet med Javaeksempelet ser du kjapt at det trengs betydelig mer kode for å kunne lage et komplett program, selv om det som skal gjøres egentlig er veldig enkelt. Dette er ikke fordi Python er *bedre* enn Java, men fordi de er lagd på ulike måter, med forskjellige bruksområder i tankene. Java er hovedsakelig ment for større prosjekter, mens Python også har hatt bittesmå programmer og prosjekter i tankene.

Dette er en av grunnene til at vi anbefaler Python, over for eksempel Java, til bruk i skolen.

2.2.2 Print-funksjonen

Å skrive ut beskjeder og informasjon til skjermen med `print` kommer til å være hovedmåten programmene våres kommuniserer med oss.

Et program kan godt inneholde flere `print`s på rad, da vil hver beskjed havne under hverandre. Vi kan også legge inn mellomrom i beskjeden for å påvirke hvor ting havner.

```
1 print('Hei')
2 print('   på')
3 print('      deg!')
```

```
Hei
   på
      deg!
```

Vi kan også lage en beskjed som går over flere linjer ved å bruke triple fnutter (''' '''), da kan vi skrive en beskjed over flere linjer. Dette kan brukes til for eksempel å skrive ut en beskjed over flere linjer, som dette diktet (*Mauren* av Inger Hagerup):

```
print('''Liten?
Jeg?
Langtifra.
Jeg er akkurat stor nok.
Fyller meg selv helt
på langs og på tvers
fra øverst til nederst.
Er du større enn deg selv kanskje?
''')
```

```
Liten?
Jeg?
Langtifra.
Jeg er akkurat stor nok.
Fyller meg selv helt
på langs og på tvers
fra øverst til nederst.
Er du større enn deg selv kanskje?
```

2.2.3 Å printe enkel grafikk med ascii-kunst

Vi kan også bruke slik flerlinjet print til å lage små tegninger ved hjelp av enkle symboler. Dette kalles for *ascii*-kunst og var en vanlig måte å lage enkel grafikk og små spill på før moderne datagrafikk kom på banen.

Her skriver vi ut en liten elefant

```
print('''
   ______/ \-.    _
.-/      (    o\_//
 |  ___   \_/\---'
 |_||  |_||
''')
```

Vår erfaring er at de fleste elever syns det er morsomt å lage enkle programmer som gjør små oppgaver og kommuniserer ved hjelp av `print`. Derimot er andre

veldig opptatt av grafikk. Her kan ascii-kunst brukes til å være litt kreativ. Å lage ascii-kunst fra bunn av kan være tidkrevende, men det finnes drøssevis man kan finne på nett om man søker på "ascii art".

2.3 Et eksempelprogram

"Hello, World"-programmet vi laget i forrige avsnitt er på mange måter det aller enkleste programmet vi kan lage. Det er derfor en god øvelse for illustrere hvordan et program skrives og kjøres, og samtidig få sjekket at nødvendig programvare er installert på maskinen man arbeider. Nå går vi igjennom et litt større eksempel for å vise hvordan et Python-program kan se ut. Alle detaljene i dette programmet vil dekkes i det neste kapitellet.

```
1  from math import pi
2
3  radius = 11   # cm
4  volum = 4*pi*radius**3/3
5
6  # Regn om fra kubikkcm til liter
7  volum /= 1000
8
9  print(f"En fotball med radius på {radius} cm")
10 print(f"har et volum på {volum:.1f} liter")
```

Dette programmet regner ut volumet av en fotball. I motsetning til "Hello, World"-eksempelet består dette programmet av flere linjer med kode. Når vi kjører dette programmet vil koden tolkes linje for linje, fra toppen og nedover. Hver kodelinje svarer til en bestemt instruks som datamaskinen vil tolke og gjennomføre. Om vi kjører programmet slik det er skrevet her får vi følgende utskrift.

```
En fotball med radius på 11 cm
har et volum på 5.6 liter
```

Vi kommer som nevnt til å dekke alt i denne koden i neste kapittel, så det er ikke nødvendig å forstå alt i detalj, men la oss prøve å tolke koden linje for linje og se om vi klarer å forstå hovedtrekkene.

I første kodelinje står det `from math import pi`, dette er en import-instruks, hvor vi importerer konstanten π. Vi trenger denne konstanten for å regne ut et vo-

lum, men Python kjenner ikke til denne fra før. Derimot kan vi, som vi gjør her, importere den fra matematikkbibloteket `math`.

Deretter definerer vi radiusen til fotballen ved å skrive `radius =11`. Her oppretter vi en variabel som heter `radius`, og setter verdien til å være 11. Vi velger denne verdien fordi en standard størrelse 5 fotball er omtrent 11 cm i radius. Vi skriver også `# cm` på denne kodelinja, men dette er faktisk ikke kode. Symbolet `#` kalles et kommentartegn, og alt som står bak dette symbolet på en kodelinje er en *kommentar*, som ikke påvirker koden. Kommentarer er et hjelpemiddel til oss som skriver koden, for lettere å holde oversikt. Python-programmet vet altså at radiusen til ballen er 11, men kjenner ikke til enheten.

I neste kodelinje regner vi ut volumet til fotballen ved å skrive ut et matematisk uttrykk som svarer til formelen for et volum (`volum =4*pi*radius**3/3`). Merk at vi her samtidig oppretter en variabel `volum`, for å ta varet på svaret av denne utregningen.

Neste linje vi har skrevet (linje 6), er en kommentarlinje. Ettersom at linja begynner med symbolet `#`, vil hele linja være en kommentar, og den påvirker ikke koden. Dette linja har vi skrevet for å gjøre det tydeligere for oss selv, og andre vi deler koden med, hvorfor vi gjør det neste steget. I neste kodelinje deler vi volumet på 1000. Dette gjør vi fordi vi oppgir radiusen i cm, slik at volumet egentlig er cm^3, men vi ønsker istedet å ha volumet oppgitt i antall liter, eller dm^3. M

Til slutt har vi to kodelinjer hvor vi skriver resultatet ut til skjermen ved hjelp av `print`-funksjonen—den samme funksjonen vi brukte i "Hello, World"-programmet vårt. I dette tilfellet er bruken av print litt mer avansert, fordi vi fletter inn verdiene av variablene `radius` og `volum` inn i teksten, men dette kommer vi tilbake til hvordan fungerer. Poenget er at vi skriver ut en beskjed som inneholder resultatet av utregningen vår på en oversiktlig måte.

Programmet som er vist her gjør altså en enkel volumberegning, skalerer svaret og skriver det ut så brukeren kan lese det. Når denne koden først er skrevet kan den enkelt kjøres på nytt. Om man for eksempel ønsker å finne volumet av en rekke ulike fotballer trenger vi kun å endre verdien av radiusen, og kjøre på nytt med ett enkelt tastetrykk, mens alt annet holdes konstant. En størrelse 3 fotball har for eksempel en radius på ca 9.5 cm, mens en størrelse 4 fotball har en radius på ca 10.2 cm. Selv om dette er en forholdsvis liten forskjell i radius, vil det ha en dramatisk effekt på volumet:

```
En fotball med radius på 9.5 cm
Har et volum på 3.6 liter
```

```
En fotball med radius på 10.2 cm
Har et volum på 4.4 liter
```

3 Variabler

Vi skal nå se på hvordan et program kan lagre, huske på og gjenbruke informasjon. Dette gjør vi ved å bruke *variabler*, som er et av de mest fundamentale konseptene i programmering.

3.1 Opprette variabler

Vi oppretter variabler, ved å bruke likhetstegn (=), vi kan for eksempel skrive:

```
1 person = 'Kari Nordmann'
2 alder = 18
```

Når vi gjør dette sier vi at vi *oppretter* eller *lager* variabler. Et annet ord man kan bruke er å si at man *definerer* en variabel.

Det vi gjør når vi oppretter en variabel, er å fortelle datamaskinen at den skal huske på en bit med informasjon. I vårt eksempel ber vi den for eksempel huske på en person, og en alder. Informasjonen lagres i minnet på datamaskinen, og vi får et navn i programmet vårt vi kan bruke til å hente informasjonen ut og bruke senere.

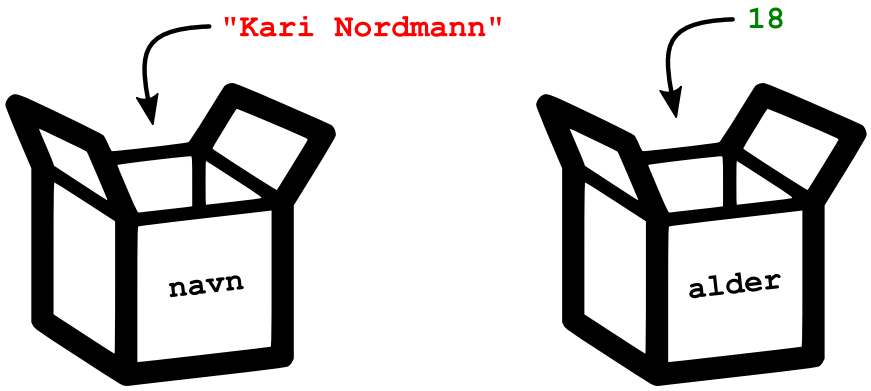
Et variabel har et *navn* og en *verdi*. For å forklare hva en variabel her så kan man godt tenke på en boks brukt til oppbevaring. Innholdet i boksen er verdien til variabelen, mens navnet skrives utenpå boksen. Datamaskinen tar seg av ansvaret med å arkivere boksen for oss i et stort lagerhus (datamaskinens minne), men kan enkelt finne dem frem igjen når vi ønsker den, takket være navnet.

3.2 Bruke variabler

Om vi har opprettet en variabel, så kan vi bruke den senere i koden vår ved å referere til den ved navn. Vi kan for eksempel skrive den ut med en `print`-kommando:

```
1 print('Hei på deg,')
2 print(person)
```

```
Hei på deg,
```



Figur 5: Variabler kan beskrives som bokser der vi oppbevarer, eller lagrer, informasjon. Variablene har en *verdi*, som er boksens innhold, og et *navn*, som brukes til å referere til og finne riktig boks.

```
Kari Nordmann
```

Når vi skriver `print(person)` er det ikke bokstavelig talt "person" som skrives ut. Fordi vi ikke bruker fnutter (') tolkes det ikke som tekst, men som kode. Det som da skjer, er at Python skjønner at 'person' er navnet på en variabel. Da går den og finner frem innholdet i den variabelen, nemlig navnet på person, og det er det som skrives ut.

Vi kan også skrive ut tekst og variabler om hverandre på én linje, da skiller vi dem bare med et komma (,) inne i parentesene:

```
1 person = 'Ola'
2 alder = 17
3
4 print('Gratulerer med dagen', person)
5 print('Du er', alder, 'år gammel idag!')
```

```
Gratulerer med dagen Ola
Du er 17 år gammel idag!
```

Se spesielt nøye på koden her. Hvilke deler av utskriften er tekst, og hvilken er variabler?

Variabler er viktige nesten uansett hva slags program vi skal lage, og det er derfor viktig å beherske begrepene vi bruker rundt variabler. Her er det altså lurt å være

å være konsekvent i bruken av at man *oppretter* variabler, og at det er et forskjell på variabelens *navn* og variabelens *verdi/innhold*. Disse begrepene blir viktig å beherske om man skal kunne snakke om kode med hverandre.

3.2.1 Flette variabler inn i tekst

Vi har sett at vi kan skrive ut variabler med `print`, og at vi kan kombinere tekster og variabler i utskrift å skille dem med komma. Vi har en annen mulighet, som er å *flette* inn variablene i teksten.

For å gjøre dette må vi gjøre to ting: Vi må skrive en `f` rett før teksten, og vi må legge på krøllparenteser (`{}`) for å si hva som er variabler:

```
navn = 'Ola'
print(f'Hei på deg {navn}!')
```

```
Hei på deg Ola!
```

Når vi skriver `f` foran teksten, så skjønner datamaskinen at `{navn}` refererer til en variabel, og den vil ta jobben med å flette variabelens innhold inn i teksten vår. Du kan huske at vi må skrive `f` foran strengen, fordi vi skal **f**lette inn variabler.

3.2.2 Eksempel: Adjektivhistorie

En enkel introduksjonsoppgave som bruker det vi har lært så langt, er å lage en enkel adjektivhistorie. Her kan vi først skrive ut en kort historie, og lage plass til forskjellige adjektiv og andre ord som skal flettes inn

```

print(f'''En {adjektiv1} høstdag i oktober
skulle den {adjektiv2}e klasse 8B på telttur
til {sted}. Alle de {adjektiv3}e elevene hadde
gledet seg til denne {adjektiv4} turen og
hadde med seg både {ting1} og {ting2}.
''')
```

Før vi faktisk kan kjøre dette programmet må vi nå gå inn og definere variablene. Hvis ikke vil Python bli veldig forvirret av at vi ber den flette inn variabler som ikke finnes!

Vi kan nå spørre om hjelp til å fullføre programmet ved å definere variablene:

```
adjektiv1 = 'rød'
adjektiv2 = 'morsom'
sted = 'månen'
adjektiv3 = 'sint'
adjektiv4 = 'rotete'
ting1 = 'tennisrekkert'
ting2 = 'øreproppper'

print(f'''En {adjektiv1} høstdag...
```

```
En rød høstdag i oktober
skulle den morsome klasse 8B på telttur
til månen. Alle de sinte elevene hadde
gledet seg til denne rotete turen og
hadde med seg både tennisrekkert og øreproppper.
```

3.3 Input

Vi har nå sett hvordan vi kan bruke `print` for å få programmet vårt til å skrive ut informasjon og beskjeder *til* oss. Men hvordan kan programmet vårt hente inn informasjon *fra* oss? Eller eventuelt fra noen andre som bruker programmet vi har skrevet? Vi kaller gjerne dette `brukeren` av programmet, det kan være oss, eller noen vi kjenner og har delt programmet med.

På engelsk kaller vi gjerne dette for "input" og "output" i programmering. Disse ordene har blitt låneord i det norske språk. Input er det et program tar inn, og output er det et program sender ut. Vi har sett at vi kan bruke `print` til å skrive beskjeder ut til brukeren, som er et eksempel på output. Nå skal vi lære om en kommando som heter `input`, som tar informasjon inn fra brukeren ved å stille spørsmål.

Å bruke `input` ligner veldig på å bruke `print`. Vi trenger parenteser, og vi kan skrive en beskjed inn mellom parentesene. Denne beskjeden skrives ut til skjermen, akkurat som med `print`. Men i tilegg til å skrive ut beskjeden, så vil programme stoppe opp etter du har brukt `input`, og vente til brukeren skriver inn et svar. Først når brukeren har skrevet inn svaret sitt og trykket *Enter*, vil programmet fortsette. La oss se på et eksempel:

```
navn = input('Hva heter du? ')

print(f'Hei {navn}. Hyggelig å møte deg!')
```

```
Hva heter du? Jonas
Hei Jonas. Hyggelig å møte deg!
```

Dette programmet spør først brukeren om hva de heter. Etter at brukeren har svart, bruker så programmet det den har lært til å skrive ut en personlig melding.

I eksempelkjøringen her i kompendiet markerer vi det brukeren skriver inn i blått. Når du selv kjører dette programmet vil programmet stoppe opp, og vi får se en markør i vinduet, slik at vi kan skrive inn svaret vårt.

Merk spesielt at vi skriver

```
navn = input('Hva heter du?')
```

Dette er fordi vi oppretter en variabel, og trenger derfor likhetstegn (=), og som alltid er navnet på variabelen på venstre side, her `navn`, og innholdet står på høyreside. I dette tilfellet blir "innholdet" på høyre side det brukeren svarer på spørsmålet som har blitt stilt.

Vi må stille spørsmålet og opprette variabelen på én kodelinje, fordi variabler er slik datamaskinen husker på informasjon. Om vi kun skriver

```
input('Hvor gammel er du?')
```

Så vil programmet stille spørsmålet, men den vil rett og slett ikke huske på svaret brukeren gir og det blir glemt med en gang.

Akkurat det med at du må opprette en variabel samtidig som du stiller spørsmålet kan være forvirrende for mange, og det er en vanlig feil å glemme å gjøre dette. I det siste eksempelet burde vi altså ha skrevet

```
alder = input('Hvor gammel er du?')
```

for å ta vare og huske på svaret fra brukeren.

3.3.1 Oppdatert adjektivhistorie

Vi kan nå gå tilbake og oppdatere adjektivhistorien vår, så den spør om nye ord hver gang programmet kjøres. Da bytter vi fra å definere variabler med gitt ord, til å spørre en bruker om dem med `input`.

```
adjektiv1 = input('Gi meg et adjektiv: ')
adjektiv2 = input('...og så et adjektiv til: ')
sted = input('...og så et sted: ')
adjektiv3 = input('...og så et adjektiv: ')
adjektiv4 = input('...et siste adjektiv: ')
ting1 = input('Nå trenger jeg en ting: ')
ting2 = input('...og så en siste ting: ')

print(f'''En {adjektiv1} høstdag...
```

Etter å ha laget dette programmet og sett at det fungerer som det skal, så kan man nå kjøre det, og få en klassekamerat eller en annen venn til å fylle ut. I noen programvarer går det an å ekspandere konsollen, slik at man ikke kan se koden, kun utskriften. Om dette er mulig kan det gjøre prosessen mer morsom for elevene.

3.4 Utregninger

Nå som vi har lært litt om å opprette og bruke variabler, la oss begynne å se litt på hvordan vi kan regne i Python. En av de store fordelene med datamaskiner er at de er så flinke til å regne. Man kan derfor godt for eksempel bruke Python som kalkulator.

La oss se på et eksempel. La oss si vi ønsker å regne ut hvor mange liter luft det er i en fotball. Om vi slår det opp ser vi at en vanlig fotball har en radius på ca 11 cm, eller 1.1 dm. Vi kan da gjøre utregningen som følger

```
pi = 3.14
radius = 1.1
volum = 4*pi*radius**3/3
```

Her definerer vi at vi skal ha en variabel, `radius` som skal være 1.1. Merk at vi ikke kan spesifisere enheten i koden, tilsvarende hvordan vi gjør det på en kalkulator. I tilegg må vi definere tallet π, da Python ikke kjenner til hva denne er. Deretter

regner vi ut volumet av fotballen ved å skrive ut et uttrykk som bruker variablene `pi` og `radius`. Merk at vi skriver `*` for å gange (asterisk/stjerne) og `**` for *opphøyd i*.

Hvis du skriver dette programmet inn på din egen maskin, og kjører det, så skjer det ingenting. Eller mer riktig, det skjer ingenting utenfor programmet, ingenting skrives ut. Dette er fordi vi ikke har brukt noen `print` kommando. Det vi istedet har gjort er å lagre resultatet i en ny variabel vi har kalt `volum`.

Om vi vil skrive ut resultatet av utregningen kan vi da skrive:

```
1 print(volum)
```

```
5.5724533333333355
```

Dette er verdien Python har regnet ut for oss. Derimot er den ikke så veldig pen, fordi den har fått fryktelig mange desimaler, langt fler enn antall gjeldende siffer.

For å få en penere utskrift kan vi bruke variabelfletting slik vi viste tidligere, men i tilegg legge til litt tileggsinfo:

```
1 print(f'Ballen rommer {volum:.1f} liter luft.')
```

```
Ballen rommer 5.6 liter luft.
```

Her skriver vi en `f` som vanlig, for å signalisere at vi skal flette. Men når vi skriver inn variabelen med krøllparanteser legger vi til et kolon (`{volum:}`), alt etter kolonet gir mer info om *hvordan* volumet skal skrives ut. Ved å skrive `.1f` sier vi at vi skal ha *en* desimal. Bokstaven står for **f**loat, eller *flyttall*, som er det vi kaller desimaltall på datamaskinen. Dette kommer vi tilbake til.

Når vi skriver kodelinjen

```
1 volum = 4*pi*radius**3/3
```

Så vil Python først regne ut det som står på høyre side av likhetstegnet. Resultatet, som blir en eller annen tallverdi, lagres så i variabelen på venstre side. Dette har to viktige konsekvenser:

1. For at utregningen skal fungere, så må `pi` og `radius` være definerte variabler.

Man kunne kanskje tenke seg at man kan bruke en "'ukjent"' for å lage en matematikkligning, og så spesifisere den senere, men det går altså ikke på denne måten.

2. Det er *resultatet* som lagres i den nye variabelen, ikke den matematiske formelen. Når `volum`-først er opprettet glemmer den hvor verdien kom fra.

   Om man så i ettertid endrer radiusen for eksempel, så vil *ikke* volumet endre seg. Da må vi isåfall gjøre volumberegningen på nytt.

Dette kan du prøve selv:

```
pi = 3.14
R = 1
V = 4*pi*R**3/3
print(f'Radius: {R} Volum: {V:.1f}')

R = 2
print(f'Radius: {R} Volum: {V:.1f}')

V = 4*pi*R**3/3
print(f'Radius: {R} Volum: {V:.1f}')
```

```
Radius: 1 Volum: 4.2
Radius: 2 Volum: 4.2
Radius: 2 Volum: 33.5
```

Her ser vi at vi i den midterste utskriften har endret radiusen, men volumet er uendret. Først når vi har utført utregningen på nytt vil volumet oppdatere seg.

3.5 Matematiske operasjoner

Vi har alle de grunnleggende matematiske operasjonene tilgjengelig i Python. Merk spesielt at for potens skriver vi `**`, og ikke `^` som i visse andre programmeringsspråk. Merk også at vi må skrive ut *ab* som `a*b`, da det ikke er implisitt multiplikasjon av variabler slik som i matematikk.

Innebygde operasjoner

Operasjon	Matematisk	Python
Addisjon	$a + b$	`a + b`
Subtraksjon	$a - b$	`a - b`
Multiplikasjon	$a \cdot b$	`a*b`
Divisjon	$\frac{a}{b}$	`a/b`
Potens	a^b	`a**b`
Heltallsdivisjon	$\lfloor a/b \rfloor$	`a//b`
Modulo	$a \mod b$	`a % b`

Av og til ønsker vi litt mer avanserte matematiske funksjoner, som for eksempel kvadratrøtter og logaritmer. Disse må vi først *importere* før vi kan bruke dem. Si for eksempel at vi ønsker å bruke kvadratrot. Dette kan vi gjøre med funksjonen `sqrt` (kort for square root) i biblioteket `math`. Så da skriver vi

```
1 from math import sqrt
```

Etter å ha importert en funksjonen kan vi bruke den fritt i resten av koden. Det er vanlig å legge slike importeringer helt i toppen av et program.

```
1 print(sqrt(81))
```

```
9.0
```

Man kan også importere *alt* som er inneholdt i et bibliotek ved å skrive stjerne:

```
1 from math import *
```

Tabellen under lister noen få av de matematiske funksjonene og operasjonene vi kan importere. Alle disse finnes i `math`.

Kan importeres

Operasjon	Matematisk	Python
Kvadratrot	$\sqrt{x}$	`sqrt(x)`
Naturlig Logaritme	$\ln x$	`log(x)`
10-er Logaritme	$\log x$	`log10(x)`
2-er Logaritme	$\log_2 x$	`log2(x)`
Sinus	$\sin x$	`sin(x)`
Eksponentialfunksjonen	e^x	`exp(x)`

I tillegg til disse operasjonene og funksjonene er det mulig å importere konstanter, som f.eks π og e. Disse kommer da med mange desimalers nøyaktighet:

```
from math import pi, e
print(pi)
print(e)
```

```
3.141592653589793
2.718281828459045
```

Vi kan derfor endre fotballeksempelet vårt over ved å istedenfor å definere π selv, så importerer vi den bare.

3.5.1 Rekkefølge og prioritet

Python følger de vanlige matematiske reglene for rekkefølge av operasjoner, for eksempel ganger den før den adderer. Vi kan også bruke parenteser for å påvirke dette:

```
a = 3 + 4/2
b = (3 + 4)/2
print(a)
print(b)
```

```
5.0
3.5
```

3.5.2 Oppdatere variabler

Etter vi har opprettet en variabel i Python, så er det mulig å oppdatere den. Da kan man enten overskrive variabelen fullstendig:

```
radius = 10
radius = 20
```

Her vil vi rett og slett erstatte innholdet i variabelen fullstendig.

En annen mulighet er å endre litt på variabelen. Si for eksempel at vi har en bankkonto og vi gjør et innskudd på 1000 kroner. Dette kan vi gjøre som følger

```
saldo = 25450
saldo += 1000
print(saldo)
```

```
26450
```

Her skriver vi `+=`. Vi skriver `+` fordi vi skal *legge til* 1000 kroner, og vi skriver `=` fordi vi skal re-definere en variabel.

På samme måte kunne vi brukt `-=` for å gjøre et uttak. Si for eksempel vi bruker bankkortet vårt til å betale en vare:

```
saldo = 18900
pris = 450
saldo -= pris
print(saldo)
```

```
18450
```

Tilsvarende kan vi også skrive `*=` for å multiplisere inn et tall, `/=` for å dele, og `**=` for å opphøye en variabel i noe.

Si for eksempel en vare i butikken koster originalt 499,-, men så blir den satt ned 30 %. Da kan vi skrive

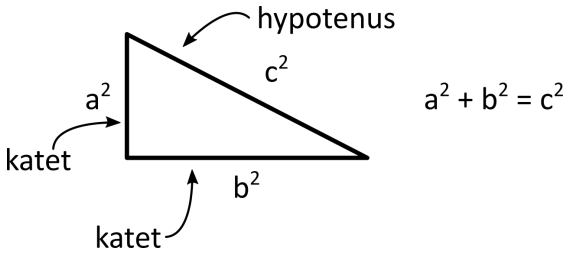
```
pris = 499
pris *= 0.7
print(f'Ny pris er {pris:.0f} kr')
```

```
Ny pris er 349 kr
```

3.6 Input og regning

Vi har nå lært litt om hvordan vi kan bruke Python til å gjøre enkle beregninger. Nå kan vi kombinere dette med `input` til å lage et program som løser et matematisk problem for oss. Derimot er det et lite problem som står i veien, som vi skal vise nå.

La oss for eksempel lage et program som regner ut hypotenusen i en trekant, når brukeren oppgir lengden på katetene. For å gjøre dette må vi bruke Pytagoras



Vi lager nå et program som spør om de to katetene, og så regner ut og skriver ut hypotenusen. For å finne c trenger vi kvadratrot, så vi må importere `sqrt`:

```
1 from math import sqrt
2
3 a = input('Hva er lengden på den første kateten? ')
4 b = input('og lengden på den andre kateten? ')
5
6 c = sqrt(a**2 + b**2)
7
8 print('Hypotenusen er {c:.1f} lang')
```

Om du prøver å kjøre dette, så vil det ikke fungere. Problemet er at nå vi bruker `input`, så tolkes svaret fra brukeren *alltid* som tekst. Så selv om brukeren skriver inn tall, så tolkes dette som tekst, og så prøver vi å regne med tekstvariabler.

La oss vise dette med et enklere eksempel. Se på denne kodesnutten:

```
a = 7
b = 8
print(f'{a} + {b} = {a + b}')
```

```
7 + 8 = 15
```

Her lager vi to *tallvariabler*, og når vi legger disse sammen med `a + b` så tolkes dette som vanlig addisjon.

Om vi derimot endrer hvordan vi definerer variablene til

```
a = '7'
b = '8'
print(f'{a} + {b} = {a + b}')
```

```
7 + 8 = 78
```

Her blir jo svaret helt feil! Dette er fordi vi her lager to *tekstvariabler*, istedet for tallvariabler. Grunnen til at det skjer at vi har med fnutter (`'`) rundt verdiene, så de tolkes som tekster. Når vi adderer tekster tror Python vi ønsker å skjøte dem sammen.

Å bruke `input` til å hente inn informasjon vil *alltid* gi tekstvariabler, selv om brukeren skriver inn et tall. Derfor må vi selv passe på å konvertere svaret til tallvariabler, om vi vet at vi skal regne med dem.

Dette kan vi gjøre ved å skrive for eksempel

```
år = 2018
fødselsår = int(input("Hvilket år ble du født? "))

print(f"Da blir du {2018-fødselsår} år i år!")
```

Her skriver vi `int()` rundt `input`-funksjonen. Dette er fordi *int* er kort for *integer* som betyr heltall på engelsk. Når vi skriver `int()` sier vi altså til Python at svaret skal tolkes som et heltall.

Om vi istedet ønsker et desimaltall, bruker vi `float()`, fordi *flyt*tall er det vi kaller desimaltall på datamaskinen.

Så vi kan gå tilbake og oppdatere Pytagorasprogrammet vårt så det fungerer som

det skal. Da må vi spesifisere at inputen skal være desimaltall, ved å legge til `float()`:

```
from math import sqrt

a = float(input('Hva er lengden på den første kateten? '))
b = float(input('og lengden på den andre kateten? '))

c = sqrt(a**2 + b**2)

print(f'Hypotenusen er {c:.1f} lang')
```

```
Hva er lengden på den første kateten? 4.5
og lengden på den andre kateten? 5.2
Hypotenusen er 6.9 lang
```

3.7 Eksempel: Muffinsoppskrift

Når man har lært de grunnleggende elementene av variabler, `input` og `print` kan man begynne å lage enkle programmer som spør om input fra brukeren, gjør en liten utregning, og så skriver ut svaret. Slike små "kalkulatorprogram" er fine øvelser for å få elever til å konkretisere og sette opp hvordan man løser små problemer. Samtidig er de meget god øving i programmering og gode måter å forklare rollen til variabler.

Et mulig program vi kan lage, er for eksempel et program som skalerer en muffinsoppskrift utifra hvor mange muffins vi ønsker å lage. Et slikt program trenger tre hovedelementer

- Vi må spørre brukeren hvor mange muffins de vil lage

- Vi må finne ut hvor mye vi trenger av hver ingrediens
- Vi må skrive ut oppskriften med riktig mengde ingredienser

Med det vi har lært så langt kan vi skrive dette programmet. Det første vi må gjøre er å spørre hvilket antall muffins som skal bli bakt. Dette har vi sett av i kan gjøre med `input`, men vi må huske at et antall muffins er et tall, så vi må også legge med `int()`

```
n = int(input("Hvor mange muffins skal du lage? ")
```

For neste steg må vi regne ut hvor mye ingredienser vi trenger. For å gjøre dette har vi funnet en muffinsoppskrift på nett og funnet ut hvor mye hver ingrediens det var.

```
print()
print("Da trenger du omtrent:")
print(f"- {10*n} gram smør.")
print(f"- {10*n} gram sukker.")
print(f"- {10*n} gram hvetemel.")
print(f"- {n/5} egg.")
print(f"- {n/10} ts bakepulver.")
```

Merk at den første linjen har en "tom" print (`print()`), dette gjør vi for å få en blank linje mellom spørsmålet til brukeren og ingrediensene. Her har vi valgt å regne ut den totale mengden ingredienser rett i `print`-funksjonen. Alternativt kunne vi opprettet nye variabler, for eksempel `mel`, `sukker`, osv.

Til slutt bør vi jo også skrive ut hva man skal gjøre med ingrediensene

```
print("""
Smelt smøret og la det kjøle seg litt ned.
Pisk eggene sammen med sukker, og rør så forsiktig
inn smøret, bakepulver og hvetemel.

Ta røra i muffinsformer og stek midt i ovnen
ved 180 grader i ca 10 minutter.""")
```

Om vi slår sammen disse tre bitene har vi et komplett program

```
Hvor mange muffins skal du lage? 12
```

```
Da trenger du omtrent:
- 120 gram smør.
- 120 gram sukker.
- 120 gram hvetemel.
- 2.4 egg.
- 1.2 ts bakepulver.

Smelt smøret og la det kjøle seg litt ned.
Pisk eggene sammen med sukker, og rør så forsiktig
inn smøret, bakepulver og hvetemel.

Ta røra i muffinsformer og stek midt i ovnen
ved 180 grader i ca 10 minutter.
```

Dette programmet fungerer akkurat som bestilt, men det er jo litt rart å få beskjed om å bruke for eksempel "2.4" egg. Her kan vi enten anta at brukeren av programmet er smart nok til å runde av litt selv, eller vi kan bygge det inn i programmet om ønskelig.

La oss si at vi ønsker å runde av til nærmeste hele egg, da kan vi bruke `round()`.

```
1 print(f"- {round(n/5)} egg.")
```

Nå vil programmet alltid si at vi skal bruke et heltallig antall egg. Det fungerer fint for et større antall egg, men om vi for eksempel sier vi skal bake bare én muffins blir det litt rart, for nå sier programmet isåfall at vi trenger 0 egg. Her kunne vi lagt til enda mer komplisert oppførsel, at det for eksempel alltid skal være minst ett egg med, eller programmet kan protestere om man prøver å bake mindre enn 4 muffins eller lignende. Vi viser ikke kode for disse løsningene her, men du kan prøve deg frem selv.

Dette er et eksempel på at et program som er tiltenkt et formål kanskje har litt rar oppførsel i visse scenarier. Som programmerer må man bestemme seg for hvordan man ønsker at et program skal oppføre seg, og det er en viktig del av prosessen å veie fordeler og ulemper å gjøre ulike designvalg.

Notis til klasserommet

Om man ønsker å bruke en lignende oppgave i klasserommet kan man la

elevene selv bruke en muffinsoppskrift fra en bok eller nettside til å finne mengden ingredienser per muffins.

4 Tester og logikk

Det siste programmeringskonseptet vi dekker i dette kurset er *tester*. Tester er viktige fordi det er slik vi innarbeider logikk i programmene våres. Da kan vi lage programmer som forgrener seg, og gjør forskjellige ting basert på omstendigheter. For eksempel kan programmet stille et spørsmål til brukeren, og så bruke svaret til å bestemme hva det skal gjøre videre. Eller kanskje programmet skal rulle en terning, og avhengig av resultatet gjøre forskjellige ting. Eller i matematikken, kanskje resultatet av en utregning påvirker hva man gjør videre i algoritmen. For eksempel ved å regne ut kvadratrøtter, avhengig av hva diskriminanten er, så må man finne ingen, 1 eller 2 røtter.

På engelsk kaller man gjerne tester for *if-tester*. Order "if" er "hvis" på norsk, så derfor kan vi kalle dem "hvis-setninger", eller eventuelt "hvis-så" setninger.

4.1 Å skrive tester i Python

Den enkleste formen for en test, er bokstavelig talt en "hvis-så". *Hvis* en betingelse er sann, *så* gjør noe. Hvis betingelsen ikke er sann, så gjør vi ikke tingen. La oss se på et eksempel i Python

```
1 svar = input("Vil du høre en vits?")
2 if svar == "ja":
3     print("Hørt om veterinæren som var dyr i drift?")
```

Her spør vi først brukeren om de vil høre en vits. Så sier vi at *hvis* svaret er ja, så skal vi fortelle vitsen. I Python skriver vi dette som

```
1 if <betingelse>:
```

Og "så"-delen gis et innrykk. All kode med innrykk skjer *kun* hvis betingelsen er sann. Om brukeren svarer noe annet enn "ja", så får de ikke høre vitsen.

En betingelse er noe som kan tolkes som må være enten sant, eller falskt. I vårt tilfelle skriver vi

```
1 svar == 'ja'
```

Det betyr at vi sammenligner to ting, hvis de to er like er det sant, hvis de er ulike er det falskt. Vi bruker to likhetstegn (==) fordi ett likhetstegn er holdt av til å

definere variabler. Merk at brukeren her må svare nøyaktig "ja" for at betingelsen skal stemme. Vi kan sjekke for flere betingelser ved å skrive `or`, altså "eller":

```
if svar == "Ja" or svar == "ja":
    print("Hørt om veterinæren som var dyr i drift?")
```

Alternativet til `or` (eller) er `and` (og).

4.1.1 Hvis-ellers

Eksempelet så langt er en ren "hvis-så". Kun hvis betingelsen er sann, så skjer noe. Men vi kan også si hva som skal skje dersom betingelsen *ikke* var sann. Dette kan vi kalle en "hvis-ellers" setning. På engelsk heter "ellers" for "else", så da skriver vi det som følger:

```
svar = input("Vil du høre en vits?")

if svar == "Ja" or svar == "ja":
    print("Hørt om veterinæren som var dyr i drift?")

else:
    print("Neivel, din surpomp")
```

Så nå vil vi *enten* fortelle en vits, *eller* kalle brukeren en surpomp. Merk at vi ikke har noen betingelse på "else"-delen av koden, den skjer bare hvis betingelsen *ikke* er sann. En betingelse er alltid enten sann eller usann.

4.1.2 Logiske operatorer

Vi kan bytte ut likhetstegnene med andre logiske operatorer, for eksempel:

```
aldersgrense = 16
alder = int(input("Hvor gammel er du?"))

if alder >= aldersgrense:
    print("Da er du er gammel nok til å se denne filmen.")
else:
    print("Da er du ikke gammel nok til å se denne filmen.
       ")
```

```
8     print("Du må ha med en voksen for å slippe inn.")
```

Her bruker vi altså `>=` for å sjekke om alderen er større eller lik aldersgrensen.

Her er en tabell over de vanligste og viktigste logiske testene:

Vanlige betingelser

Matematisk symbol	Kode	Tolkning
$a < b$	`a < b`	Mindre enn
$a > b$	`a > b`	Større enn
$a = b$	`a == b`	Lik
$a \leq b$	`a <= b`	Mindre eller lik
$a \geq b$	`a >= b`	Større eller lik
$a \neq b$	`a != b`	Ikke lik

Merk at hvordan *større enn* og *mindre enn* tolkes avhengig 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 sorterte dem alfabetisk. Dette er viktig med tanke på `input`, for om vi skal sammenligne `a` og `b` som tall må vi gjøre dem om til tallvariabler.

I tillegg til disse vanlige betingelsene kan vi også legge til ordet `not`, om vi ønsker å invertere betingelsen. Vi kan for eksempel skrive `if not a ==b`. I tilegg kan vi bruke `and` og `or` for å kombinere to eller flere betingelser. Forskjellen er at om vi bruker `and` vil testen bestå kun om begge betingelsene er sanne, mens om vi bruker *or* trenger bare en av betingelsene å være sanne for at testen passerer.

Disse betingelsene og måtene å kombinere dem på kalles gjerne Boolsk logikk, etter matematikeren 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.

4.2 Eksempel: Quiz

Ved å bruke if-else tester kan vi lage en kort quiz:

```
riktige = 0

svar = input("Hva heter hovedstaden i Portugal? ")
if svar == "Lisboa":
    print("Det stemmer!")
    riktige += 1
else:
    print("Det er feil. Riktig svar er Lisboa.")

svar = input("Hva slags dyr er Scooby Doo? ")
if svar == "hund":
    print("Det stemmer!")
    riktige += 1
else:
    print("Det er feil. Scooby Doo er en hund.")

print(f"Det var det! Du fikk til {riktige} av 2 spørsmål."
   )

if riktige == 2:
    print("Veldig bra jobba!")
```

Dette er et kort eksempel med bare to spørsmål, men her kan man enkelt utvide quizzen og legge til flere spørsmål. For hvert spørsmål sjekker vi svaret og skriver eventuelt ut riktig svar om brukeren tar feil. I tilegg har vi opprettet en variabel for å holde styr på antall riktige svar.

En utfordring med en slik oppgave er at brukeren må skrive inn nøyaktig riktig svar, for at de skal få riktig. Det finnes små triks vi kan gjøre for å forbedre dette litt, men det er overraskende vanskelig å lage et program som godtar mange forskjellige versjoner av samme svar.

4.3 Mer enn to utfall

Av og til trenger vi en test der det ikke er enten-eller, men at vi har mer en to mulige utfall. Dette kan vi lage i Python med nøkkelordet `elif`. Ordet "elif" er ikke et ekte ord på engelsk, men en sammenslåing av ordene "else if". Det betyr altså at dersom den første betingelsen ikke er sann, så kan vi sjekke en ny.

Si for eksempel at vi har lagd et spill der to spillere har samlet opp poeng. På slutten må vi kåre en vinner. Da er det tre mulige utfall: Spiller A slår spiller B, eller så slår spiller B spiller A, eller så blir det uavgjort. Det kan vi skrive som følger:

```
poeng_a = 290
poeng_b = 320

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

Her sjekkes først den første testen. Dersom den er sann, så 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. Uansett så vil kun ett av utfallene skje.

Det er ingen begrensning på hvor mange `elif` vi kan bruke. Vi kan for eksempel lage et program som kaster en terning, og har 6 ulike utfall basert på resultatet.

For å skjønne forskjellen på en ren "hvis-så", en "hvis-ellers" og en "hvis-ellers hvis-ellers", så kan det lønne seg å tegne dem opp.

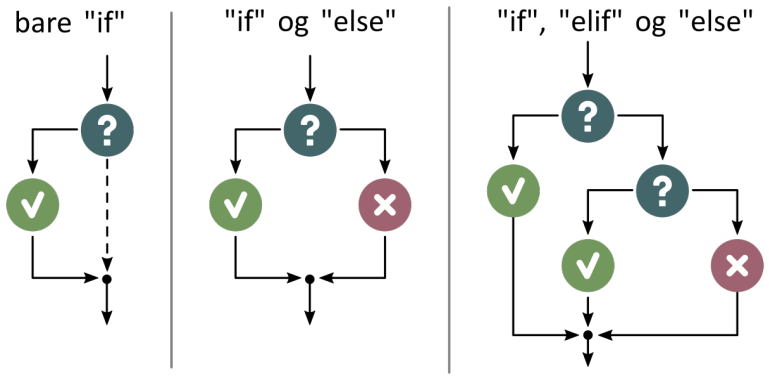
4.4 Eksempelprosjekt: Choose your own adventure

En egen kategori av spill kalles "Choose your own adventure"-spill. Disse er gjerne gamle tekstbaserte spill, fra før avansert datamaskiner og grafikkmuligheter var tilgjengelige. For eksempel *Adventure* (1976) og *Zork* (1977).

Disse spillene er laget på ikke mye mer en printing, input og if-tester.

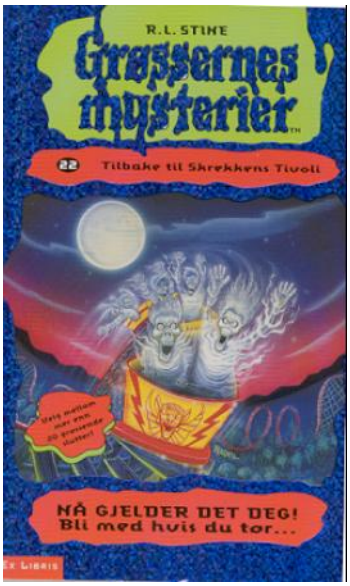
Spillene er en historie som fortelles deler av gangen. Av og til vil spillene stoppe opp og spørre brukeren hva de velger å gjøre. På den måten kan spillet forgrene seg.

Elever kan lage enkle, tekstbaserte spill selv. Disse blir gjerne korte, men kan fortsatt være gøy. Om man tar feil valg så ender kanskje historien brått, så for å



Figur 6: Tre forskjellige typer tester. Til venstre finner vi en hvis-så setning, om betingelsen ikke er sann hopper vi bare over hele greia. I midten er en hvis-ellers, her gjør vi to ulike ting avhengig av betingelsen. Til slutt har vi en hvis-ellers hvis-ellers, denne kan bestå av så mange ulike forgreninger vi måtte ønsker.

klare å komme til slutten av spillet må man gjøre riktige valg hele veien.



Figur 7: *Grøsserne Mysterier* var *Choose your own adventure* historier i bokformat. Her kom man til veivalg, der man skulle bytte til forskjellige sider i boka avhengig av hvilke valg man tok. Om man gjorde feil valg endte ofte historien på skrekkelige og skumle vis.

4.5 Eksempel: Flere typer muffins

I avsnitt 3.7 viste vi et eksempel på et program som skalerte en muffinsoppskrift utifra hvor mange muffins brukeren ønsket å lage. La oss nå inkludere tester i dette programmet for å gjøre det litt mer avansert.

Sist hadde vi kun én muffinsoppskrift, men det finnes jo mange ulike oppskrifter for muffins! Derfor utvider vi nå så vi har ulike oppskrifter brukeren kan velge fra.

Vi begynner med å stille brukeren to spørsmål, hvor mange muffins de vil lage, og hva slags muffins de vil lage. Vi gir dem 4 alternativer

```
n = int(input("Hvor mange muffins skal du lage? "))

print("""
Muffinstyper:
1. Vaniljemuffins
2. Vaniljemuffins med sjokoladebiter
3. Sjokolademuffins
4. Blåbærmuffins
""")
muffinsslag = input("Hvilken type vil du lage? (Skriv inn
   tallet som svarer til ditt valg.) ")
```

For å gjøre programmet mer "idiotsikkert" sier vi her at brukeren skal skrive inn et tall for å gjøre valget sitt. Alternativt kunne de skrevet inn navnet på muffinsen, men da blir det fort surr på grunn av skrivefeil og lignende.

For alle muffinstypene har vi samme grunnoppskrift, så denne kan vi skrive ut uansett. Men deretter har vi en if-test som skriver ut de ekstraingrediensene vi trenger

```
print()
print("Da trenger du omtrent:")
print(f"- {10*n} gram smør.")
print(f"- {10*n} gram sukker.")
print(f"- {10*n} gram hvetemel.")
print(f"- {round(n/5)} egg.")
```

```
print(f"- {n/10} ts bakepulver.")

if muffinsslag == "1":
    print(f"- {n/10} ts vaniljesukker")
elif muffinsslag == "2":
    print(f"- {n/10} ts vaniljesukker")
    print(f"- {5*n} gram hakket sjokolade")
elif muffinsslag == "3":
    print(f"- {n/10} ts kakao")
elif muffinsslag == "4":
    print(f"- {20*n} gram ferske eller fryste blåbær")
```

Vi bruker samme fremgangsmåte på selve oppskriften, det meste er felles, så vi skriver det ut først, og til slutt litt ekstra der det trengs

```
print("""
Smelt smøret og la det kjøle seg litt ned.
Pisk eggene sammen med sukker, og rør så forsiktig
inn smøret og de tørre ingrediensene.""")

if muffinsslag == 2:
    print("Bland til slutt inn den hakkede sjokoladen.")
elif muffinsslag == "4":
    print("Vend så blåbærene forsiktig inn i røra.")

print("""
Ta røra i muffinsformer og stek midt i ovnen
ved 180 grader i ca 10 minutter.""")
```

Vi har nå lagd et program som justerer oppskriften både etter antall muffins, men også hva slags muffins man skal lage. Vi har brukt både variabler og logikk for å få til dette. Derimot har vi ikke laget et veldig "idiotsikkert"-program, for hva skjer om brukeren skriver inn at de vil ha "sjokolademuffins" for eksempel? Se om du kan finne en måte å gjøre programmet mer robust.

Programmet vi her har skissert ut er forholdsvis stort, på ca 40 kodelinjer. For en nybegynner vil det være en altfor stor oppgave å skrive ut et helt slikt program i én stor jafs. Istedet er det lurt å først fokusere på den enklere muffinsoppgaven skissert i avsnitt 3.7, og deretter utvide programmet stegvis. Først kan man fokusere på kun to muffinstyper, og så legge til fler, osv.

Når man skal skrive programmer som er mer enn et par kodelinjer er det alltid viktig å teste koden underveis, ellers blir det utrolig vanskelig å finne feilene som nesten garantert vil oppstå.

Et alternativ er også at elevene selv programmerer den enkle muffinsoppgaven, og så programmerer læreren den utvidede med tester i plenum, etter innspill fra elevene.

5 Løkker

Løkker brukes for å gjenta en prosess mange ganger. Dette høres kanskje ikke så nyttig ut, men er det av de viktigste konseptene innen programmering. Den ene grunnen er at vi kan spare oss selv arbeid, ved å skrive en kort kode som datamaskinen gjennomfører mange ganger, istedenfor å måtte skrive ut alle instruksene gang på gang. Den andre grunnen er at veldig mange *algoritmer* lages med løkker, det vil si at løkker lar oss løse viktige problemstillinger, noe vi kommer litt tilbake til senere.

Det er vanlig å snakke om to forskjellige typer løkker i programmering, og disse kalles gjerne *for*-løkker og *while*-løkker på englesk. I bunn og grunn er de to veldig like, og går begge ut på å gjenta en prosess mange ganger.

5.1 Historie: Shampoo-algoritmen

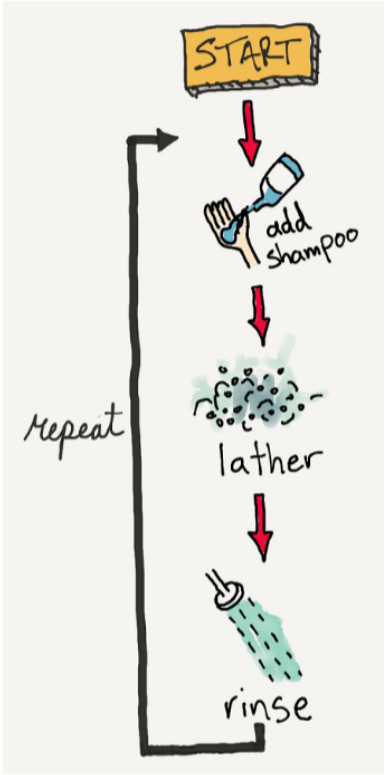
Som en introduksjon til løkker, eller bare som et humoristisk innslag, kan man dele historien om *sjampoalgoritmen* med elevene. Sjampoalgoritmen stammer fra instrukser som man finner på mange sjampo-flasker. På engelsk er disse ofte:

- Lather
- Rinse
- Repeat

Ordet *lather* betyr å skumme opp, og betyr altså å spre sjampoen godt ut i håret, *rinse* betyr å skylle ut, og *repeat* betyr å gjenta.

Her er det spesielt det siste ordet som er litt interessant. Den siste instruksen er å *gjenta*. Om vi skal følge denne instruksen bokstavelig betyr jo det at vi begynner på nytt. Først tar vi ny sjampo i håret, så vasker vi det ut på nytt, og så til slutt: så *gjentar* vi på nytt. Dette er et eksempel på en løkke, faktisk på en *uendelig* løkke. Om man følger instruksjonene bokstavelig, så vil man stå der og vaske håret sitt for all tid, ihvertfall til man går tom for sjampo.

Sjampoalgoritmen er blitt en populær måte å introdusere konseptet algoritme, eller løkker som handler om gjentagelser. Eller bare som et eksempel på at man må være litt forsiktig når man programmerer, da en datamaskin gjerne følger instruksjoner



bokstavelig. Det brukes også gjerne som en spøk. Selve sjampoalgoritmen er så velkjent i engelsktalende land at begrepet "rinse-repeat" brukes gjerne som et idiom på å gjenta noe, selv om det ikke har noe med *skylling* å gjøre i det heletatt.

Did you hear about the programmer who got stuck in the shower? The instructions said: Lather, rinse, repeat

5.2 Løkke over tallrekker

Kanskje de aller enkleste løkkene i Python er også kanskje de viktigste i matematikken, nemlig å løkke over en tallrekke. På englesk kalles en tallrekke for en

"range", og vi bruker derfor funksjonen `range()` for å løkke over en tallrekke.

For å lage en løkke over tallrekka 1, 2, ..., 10 i Python, så skriver vi

```
for tall in range(1, 11):
    print(tall)
```

```
1
2
...
10
```

Vi begynner å lage løkka med å skrive `for tall in`. Her er ordene `for` og `in` bestemte nøkkelord i Python, og de må alltid være de samme. Ordet `tall` derimot, velger vi selv, og dette er det vi kaller for *tellevariabelen* eller *løkkevariabelen* vår. Dette er en variabel som endrer seg hver gang løkka gjentar seg selv.

Til slutt skriver vi `range(1, 11)`, dette betyr at vi ønsker tallrekka *fra og med*, til (men ikke med) 11. Det at det siste tallet ikke blir med i rekka er litt forvirrende. Det finnes gode grunner til at det er slik, men det er ikke så viktig akkurat hvorfor det er sånn. Her er det bare slik at de som lagde Python bestemte seg for at sånn skulle det fungere, og det må vi bare lære oss og huske på.

Til slutt, etter å ha skrevet `for tall in range(1, 11)` har vi et kolon (:). Hele linja kan da leses rett ut egentlig. Ordet "in" er jo engelsk, men kan byttes ut med det norske ordet "i". Altså kan vi lese løkka som at vi skriver:

> **for** hvert tall **i** tallrekka fra og med 1 til 11:

Etter denne linjen, så gir man neste kodelinje et lite *innrykk*. Det betyr at koden flyttes litt inn på linjen sin. Alle kodelinjer med et slikt innrykk hører nå til løkken, og vil gjentas hver gang løkka gjentas. I vårt eksempel er det kun én sånn kodelinje: `print(tall)`. Dette betyr at løkka vil gjenta denne kodelinja for hvert tall i tallrekka vår, og så skrive det ut.

Når vi begynner å skrive en liste vil editoren av og til lage et innrykk for oss automatisk. Om vi vil lage innrykk selv gjør Tab-knappen det for oss. Dette er knappen over Caps Lock, på venstre side av tastaturet.

5.2.1 Eksempel: Kvadrattall og kubikktall

La oss se på et annet eksempel. Et kvadrattall er et heltall som er kvadrert. La oss si vi ønsker å se litt nærmere på kvadrattall. Da kan vi lage en løkke som skriver ut kvadrattallene $1^2, \ldots, 5^2$ som følger:

```
for tall in range(1, 6):
    kvadrat = tall**2
    print(kvadrat)
```

```
1
4
9
16
25
```

Om vi endrer fra opphøyd i annen (**2) til opphøyd i tredje (**3) ville vi istedet skrevet ut kubikktall. Vi kan for eksempel lage en løkke som skriver ut begge deler:

```
for tall in range(1, 6):
    kvadrat = tall**2
    kubikk = tall**3
    print(tall, kvadrat, kubikk)
```

```
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
```

Dette fungerer fint, for hvert tall i tallrekka regner løkka ut kvadrattallet og kubikktallet og printer alle tre. Derimot blir utskriften litt rotete, fordi antall siffer endrer seg etterhvert som tallene vokser. Dette blir spesielt åpenbart om du lar tallene vokse enda lengre.

Vi kan gjøre det hele penere, ved å bruke flettestrenger:

```
for tall in range(1, 6):
    kvadrat = tall**2
    kubikk = tall**3
    print(f'{tall:4} {kvadrat:4} {kubikk:4}')
```

Her skriver vi `:4` bak variablene for å spesifisere at uavhengig av størrelsen på tallet, så skal utskriften bruke 4 plasser bredde. Dette gjør at vi får fine kolonner i utskriften, uavhengig av hvor store tallene blir.

```
   1    1    1
   2    4    8
   3    9   27
   4   16   64
   5   25  125
```

5.2.2 Sum av en tallrekke

Vi kan også bruke løkker til å regne ut summer. La oss for eksempel si vi ønsker å regne ut summen av tallene $1, 2, \ldots 10$. Det kan vi gjøre ved å løkke over denne tallrekka, og addere dem til en total, én etter én:

```
1 total = 0
2
3 for tall in range(1, 11):
4     total += tall
5
6 print(total)
```

```
55
```

Her er det to ting å merke seg spesielt. Først så bruker vi `+=` her, altså oppdatere vi totalen for hver iterasjon. Det andre er at den siste linja er *utenfor* løkka, ettersom at den ikke har fått innrykk. Altså vil løkka gjøre seg helt ferdig før programmet går videre, og vi skriver ut resultatet vårt. Summen av tallene 1 til 10 er et *trekanttall*, og dette kommer vi tilbake i oppgavene.

5.2.3 Eksempel: Fakultet

Fakultet er en matematisk operasjon som er viktig i kombinatorikk. Vi skriver fakultet med et utropstegn i matematikken. For eksempel er

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120.$$

Altså ganger vi tallrekka nedover til og med 1.

La oss lage et kort program, som regner ut fakultet for et gitt tall

```
n = int(input('Gi meg et tall: '))

total = 1
for tall in range(1, n+1):
    total *= tall

print(f'{n}! = {total}')
```

```
Gi meg et tall: 5
5! = 120
```

```
Gi meg et tall: 10
10! = 3628800
```

Vi ser at fakultet vokser veldig fort for større tall. Faktisk vokser den raskere og raskere.

Fakultet er viktig i kombinatorikk, fordi $n!$ er antall måter vi kan arrangere n ting. Ta for eksempel en vanlig kortstokk, som består av 52 kort (vi ser bortifra jokere). Hvert kort er unikt, så det er 52! måter en kortstokk kan være arrangert på. Ved å bruke programmet vi lagde kan vi se at dette svarer til

$$\begin{aligned} 52! = & 80658175170943878 57... \\ & 16606368564037669 75... \\ & 28950544088327782 40... \\ & 00000000000, \end{aligned}$$

et helt enormt tall! Om vi endrer `print`-kommandoen vår til `otal:gt`, så vil den skrive ut store tall på standardformat. Da får vi isåfall:

```
Gi meg et tall: 52
52! = 8.06582e+67
```

Som svarer til 8×10^{67}. Det er et så stort tall at det er nesten umulig å forklare det. Hver gang en kortstokk stokkes (skikkelig) velges én av disse rekkefølgene (eller *permutasjoner* som de gjerne kalles) tilfeldig. Det betyr at vi med sikkerhet kan si at når man stokker en kortstokk skikkelig, vil den bli annerledes fra alle andre stokkinger som noen sinne har vært i historien.

5.2.4 Finkontrol på tallrekka

Alle eksemplene så langt har vi brukt `range` med to tall, nemlig `range(fra, til)`. Vi kan derimot også bare oppgi ett tall, for eksempel

```
for tall in range(5):
```

Isåfall tolkes dette som tallrekka

$$0, 1, 2, 3, 4.$$

Å oppgi bare ett tall betyr altså det samme som å si at vi vil starte på 0, og gå opp til, men ikke det tallet man oppgir.

Vi kan også oppgi tre tall. Da vil de første to tolkes som vanlig: fra-til, og det siste sier noe om hvor store steg vi vil ta. Vi kan for eksempel bare løkke over partall ved å skrive:

```
for partall in range(0, 12, 2):
    print(partall)
```

```
0
2
4
6
8
10
```

Merk at 10 blir den siste utskriften, siden vi går *til, men ikke med* 12.

5.3 Eksempel: Renteberegninger

Eirik har nettopp hatt konfirmasjon, og fått mange hyggelige gaver. Nå sitter han på 10 tusen kroner. Eirik blir anbefalt å sette penge på sparekonto, slik at han kan bruke dem til å kjøpe bolig når han blir eldre.

Eirik lurer på hvor mye fortjeneste han vil få fra å ha pengene på en sparekonto, og skriver derfor et lite Pythonprogram for å finne ut dette.

La oss si at Eirik får en årlig rente på 3.45 % på en BSU konto. Om han setter inn sine 10 000 kroner, hvordan vil disse vokse over tid? Dette gjøres enklest med en

løkke, fordi vi for vært år skal gjenta de samme stegene: Først beregne rente, og så skrive ut fortjenesten.

```
penger = 10000
rente = 1.0345

for år in range(1, 11):
    penger *= rente
    print(f'Etter {år} år er det {penger} kr.')
```

```
Etter 1 år er det 10345.0 kr.
Etter 2 år er det 10701.9025 kr.
Etter 3 år er det 11071.11813625 kr.
...
```

Programmet fungerer helt fint, men selve utskriften blir rotete, fordi vi får med masse desimaler som ikke er viktige. Vi kan runde av til nærmeste desimal ved å skrive `penger:.0f` i f-printen vår, her betyr `.0` at vi skal ha null desimaler, og `f` betyr at det er en `float`, altså desimaltall. I tilegg skriver vi `{år:2}`, slik at antall år tar samme bredde, og tallene havner pent under hverandre.

```
penger = 10000
rente = 1.0345

for år in range(1, 11):
    penger *= rente
    print(f'Etter {år:2} år er det {penger:.0f}  kr.')
```

```
Etter  1 år er det 10345 kr.
Etter  2 år er det 10702 kr.
Etter  3 år er det 11071 kr.
Etter  4 år er det 11453 kr.
Etter  5 år er det 11848 kr.
Etter  6 år er det 12257 kr.
Etter  7 år er det 12680 kr.
Etter  8 år er det 13117 kr.
Etter  9 år er det 13570 kr.
Etter 10 år er det 14038 kr.
```

Dette eksempelet viser hvor god fortjeneste man kan ha på sparing, spesielt når man tenker på rentesrente. Man kan lage en mer utvidet oppgave, eller et helt

opplegg på dette der man for eksempel kan se på forskjellige måter å spare på. Eller man kan se på forskjellige lån, og hvor lang tid det tar å nedbetale dem med faste avdrag. Her kan man f.eks se på hvor forferdelige forbrukslån kan være.

5.4 Å gjenta kode n ganger

Eksemplene vi har brukt så langt har handlet om å løkke over tallrekker, og bruke disse tallrekkene til ting. Dette er en naturlig bruk av løkker, men av og til ønsker man kun å gjenta en gitt kode et gitt antall ganger, og det er uviktig å faktisk ha noen tallrekke. I disse tilfellene er det vanlig å bare bruke `range(n)`. Selv om dette teknisk sett betyr tallrekka $0, 1, \ldots n-1$, så trenger vi rett og slett ikke å bruke disse tallene til noe.

```
for tall in range(100):
    print('Spam!')
```

Her gjentar vi en utskrift hundre ganger på rad. Løkkevariabelen `tall` bruker vi aldri.

Det noen liker å gjøre når de treffer på tilfeller som dette, er å isteden skrive

```
for _ in range(100):
    print('Spam!')
```

Altså at de skriver understrek (_) istedenfor å definere en løkkevariabel. Dette gjør det tydeligere at variabelen ikke brukes til noe. Men dette kan kanskje være ekstra forvirrende for noen, så det er ikke nødvendig å nevne det om det ikke blir relevant.

5.5 Eksempel: Fiskebollesangen

Et vanlig eksempel å bruke når man dekker løkker for første gang i programmering, er drikkevisa 99 bottles of beer. Dette er en repetitiv vise der hvert vers kun bytter ut antall flasker som er igjen:

> *99 bottles of beer on the wall, 99 bottles of beer.*
> *Take one down and pass it around, 98 bottles of beer on the wall.*

Og slik fortsetter sangen til det er helt tomt for øl. Poenget er at denne sangen følger et veldig enkelt repeterende mønster, men er fortsatt slitsom å skulle synge, eller skrive ut i sin helhet. Derimot kan vi bruke en `for`-løkke til å gjøre det kjapt og effektivt.

Nå er nok drikkeviser om øl ikke midt i blinken for ungdomsskoleundervisninga, men vi kan jo isteden bruke den norske varianten om fiskebolla som lengter etter havet.

```
for vers in range(1, 101):
    print(f'''Fiskebolla lengter etter havet,
for havet er fiskebollas hjem,
dette er det {vers}. verset,
da er det bare {100-vers} igjen,
Bom-bom-bom!
''')
```

```
Fiskebollen lengter etter havet,
for havet er fiskebollens hjem,
dette er det 1. verset,
da er det bare 99 igjen,
Bom-bom-bom!

Fiskebollen lengter etter havet,
for havet er fiskebollens hjem,
dette er det 2. verset,
da er det bare 98 igjen,
Bom-bom-bom!

Fiskebolla lengter etter havet,
for havet er fiskebollas hjem,
dette er det 3. verset,
da er det bare 97 igjen,
Bom-bom-bom!

...

Fiskebolla lengter etter havet,
for havet er fiskebollas hjem,
dette er det 100. verset,
da er det bare 0 igjen,
Bom-bom-bom!
```

Dette eksempelet illustrert litt hvordan løkker kan brukes for å repetere steg, uten å koble det opp mot matematikken.

5.6 While-løkker

I eksemplene over brukte vi `for`-løkker. I andre tilfeller virker det mer naturlig å bruke `while`-løkker. Det er ikke bare i Python at man gjerne snakker om for- og `while`-løkker, det går igjen i de fleste programmeringsspråk.

En `while`-løkke minner litt om en `if`-test, i det at vi skriver en betingelse som sjekkes. I likhet med testen vil innholdet i løkka kun kjøres dersom betingelsen er sann, om betingelsen er usann, så hopper vi glatt over innholdet. Forskjellen fra `if`-testen er derimot at etter innholdet er kjørt, så sjekkes betingelsen på nytt. Dersom betingelsen *fortsatt* er sann, så kjører vi innholdet på nytt. En `while`-løkke er altså som en `if`-test som repeterer seg helt til den blir usann.

La oss se på et konkret eksempel

```
1 teller = 0
2 while teller < 10:
3     teller += 1
4     print(teller)
```

```
1
2
3
...
10
```

I denne koden lager vi først en variabel `teller`, som begynner på 0. Så lager vi en `while` løkke, der vi sjekker betingelsen `teller < 10`. Ettersom at 0 er mindre enn 10 går vi da inn i løkka og utfører koden der. Nå vokser `teller` fra 0 til 1. Så skriver vi ut variabelen, så vi får 1 som første utskrift. Etter dette sjekkes betingelsen på nytt. Nå er teller blitt til 1, men 1 er fortsatt mindre enn 10, og løkka kjøres derfor på nytt. Dette betyr at variabelen økes til 2, og skrives ut.

Dette gjentar seg helt til telleren er blitt 10. Nå vil betingelsen igjen spørre `teller < 10`, men siden 10 ikke er mindre enn 10, så vil løkka ikke kjøres.

Dette eksempelet er kanskje ikke så altfor spennende, så la oss se på et litt mer spennende eksempel.

5.7 Eksempel: Høyrentekonto

Når vi så på forløkker så vi på et eksempel der vi regnet ut hvor mye penger som var på en sparekonto for hvert år som gikk, når vi begynte med en viss sum. Med for-løkken må vi bestemme oss for hvor mange år vi skal fortsette å regne renter mens vi skriver koden.

Nå snur vi problemet på hodet og stiller spørsmålet:

- Hvor mange år må vi vente før vi har 20000 på konto?

Selve renteberegningen for hvert år blir helt lik. Men poenget er at når spørsmålet stilles på denne måten så vet vi på forhånd ikke hvor mange år vi må fortsette å regne. Dette er et perfekt eksempel på hvor en while-løkke kan være lurt.

```
penger = 10000
år = 0
rente = 1.035

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

print(f'Du må vente i {år} år.')
print(f'Det er da {penger} på konto.')
```

```
Du må vente i 21 år.
Det er da 20594 kr på konto.
```

5.8 Eksempel: Quiz

Et annet lite eksempel vi kan bruke for å illustrere styrken av en while-løkke er for å lage en quiz der brukeren kan prøve på nytt helt til de treffer.

```
svar = input("Hva heter hovedstaden i Portugal? ")

while svar != 'Lisboa':
    svar = input("Nei, det var feil. Prøv igjen. ")

print('Stemmer!')
```

Her kan det være en god idé å stoppe å tenke på hvorfor den siste `print`-kommandoen kan stå helt uavhengig av løkka.

```
Hva heter hovedstaden i Portugal? Madrid
Nei, det var feil. Prøv igjen. Porto
Nei, det var feil. Prøv igjen. Lisboa
Stemmer!
```

5.9 For- eller while-løkker?

Vi har nå vist noen korte eksempler på begge typene med løkker i Python. De to er noe forskjellig, men ikke voldsomt. Det viktigste elementet er at de begge er løkker, som betyr at de repeterer kode.

`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. Dette prøvde vi å illustrere med eksempelet med høyrentekontoen.

Noen problemer egner seg altså best til `while`-løkker, mens andre egner seg bedre til `for`-løkker. Derimot er det faktisk sånn at alle problemer som kan løses med den ene typen kan også løses med den andre. I praksis handler altså mest om hvordan man liker og tenke og personlig preferanse.

Om man har tid til å dekke begge typene med løkker i undervisningen, så er det selvfølgelig fint, men det er ikke noe krav fra kompetansemålene sin side—der står det kun at elevene skal kunne lage programmer som benytter seg av løkker.

Det er også noe debatt om hvilke løkker det er lurt å lære bort først og hvilke som er mest intuitive. Noen mener `for`-løkkene er forholdsvis rett frem, siden det er veldig konkret hvilken tallerekke de går igjennom. Mens andre mener `while`-løkkene er en god introduksjon til løkker, ettersom at de springer naturlig ut fra

`if`-testene. Dette må du gjøre deg opp dine egne tanker om, og hvordan du selv ønsker å introdusere løkker til dine elever. Uansett valg anbefaler vi å starte med løkker frakoblet.

5.10 Uendelige løkker

Med `for`-løkkene er det konkret hva man løkker over når man skriver løkka. Om man jobber med tallrekker for eksempel, har man spesifisert akkurat hvor rekka skal stoppe.

Med `while`-løkker derimot, er det teknisk sett mulig å ende opp med en løkke som *aldri* slutter. Dette gjør man ofte fordi man har rett og slett gjort en feil. La oss gjenta vårt første eksempel

```
1 teller = 0
2 while teller < 10:
3     print(teller)
```

Her har vi gjentatt koden vi hadde over, men utelatt en viktig kodelinje. Hva skjer om vi kjører denne koden? Jo, telleren skrives ut, som er `0`, så sjekkes betingelsen på nytt `0 < 10` er fortsatt sant så vi gjentar løkka. Der er altså slik at variabelen aldri økes! Vi bare skriver ut 0 gang på gang på gang.

Når vi får en situasjon der vi er inne i en uendelig løkke må vi selv ta ansvaret for å stoppe kjøringen av programmet, for dette skjønner ikke datamaskinen av seg selv.

Det å ved uhell lage kode som bare går i ring "for alltid" er noe selv erfarne programmerere kan gjøre fra tid til annen, og feil av denne typen kan være én grunn til at apper og programmer av og til henger seg og må lukkes. Slike feil kan derimot også være mye annet. Det er derfor nyttig at man prøver å selv "leke datamaskin" og gå igjennom en løkke for hånd for å være sikker på at den gjør det den skal og stopper når du forventer.

6 Sammensatt eksempel 1: FizzBuzz

FizzBuzz er en enkel lek man kan leke i par, eller i større grupper. Denne leken er ment for å lære barn om divisjon og gangetabellen, og brukes gjerne i grunnskolen.

Leken FizzBuzz har egentlig ingenting med programmering å gjøre. Derimot har det å kode opp denne leken som et dataprogram blitt til en veldig populær øvelse. Grunnen til at nettopp denne øvelsen er blitt så populær er nok at selve oppgaven er såpass lett å forstå seg på, men for å kunne kode en løsning må man beherske både løkker og tester.

Selve oppgaven er ikke så altfor komplisert, men det kan være en ordentlig utfordring for en nybegynner. Derfor har nettopp FizzBuzz blitt spesielt populær å bruke i jobbintervjuer, for å sjekke om en kandidat faktisk kan programmere eller ikke.

6.1 FizzBuzz frakoblet

La oss først forklare hvordan man leker FizzBuzz helt frakoblet fra datamaskinen. Leken fungerer best med en litt større gruppe. Leken går ut på at man skal telle oppover som en gruppe. Så en person starter og sier 1, deretter sier neste i løkka 2, så sier nestemann 3 og så videre. Utfordringen kommer av at hver gang man møter på et tall som er delelig med 3, så skal man si *Fizz*, istedenfor tallet. Når man møter på et tall som er delelig på 5, så skal man si Buzz istedenfor tallet. Så da blir rekka som følger:

$$1, 2, \text{Fizz}, 4, \text{Buzz}, \text{Fizz}, 7, \ldots$$

Om man kommer til et tall som er delelig med *både* 3 og 5, for eksempel 15, så skal man si begge deler, altså *FizzBuzz*.

Etterhvert som man teller oppover bør læreren følge med på at alle gjør riktig. Om noen bommer, eller bruker for lang tid, så avslutter man, og begynner fra starten. Målet er å komme helt til 100 uten å gjøre en eneste feil, og det skal gjerne gå litt fort! Eventuelt kan man gjøre det til en konkurranse, der hver gang noen gjør feil, så går de ut, så fortsetter man helt til man står igjen med én (eller flere) vinnere.

Om du ønsker å lage et opplegg rundt FizzBuzz med dine elever anbefaler vi å sette av nok tid til å forklare og leke leken frakoblet til at elevene skjønner oppgaven, før man setter igang med kodingen. Her kan man enten sette elevene sammen i par, eller hele klassen kan stelle seg i en ring og leke sammen. Her kan man sette av et kvarter i starten av timen for eksempel.

6.2 Å kode opp FizzBuzz

Nå ønsker vi å gå over til datamaskin og kode opp FizzBuzz. Med dette mener vi at vi ønsker å lage et program som skriver ut det man ville sagt i leken. Vi skal altså lage et program som teller oppover fra 1 til 100, men som følger disse spesielle tilleggsreglene.

Hver gang vi programmerer, spesielt når vi skal håndtere litt større, kompliserte problemer, er det viktig å gå frem i små steg. Det betyr at hver gang vi skriver litt kode, så bør vi sjekke at koden vi har skrevet gjør det vi tror den gjør.

I programmeringen kan de fleste problemer løses på mange måter. Vi vil nå gå igjennom å vise *én* mulig måte å jobbe seg igjennom dette problemet. Men det er fullt mulig å gå frem på alternativer måter.

Et naturlig første steg er å ignorere tilleggsregler våres fullstendig, å lage et program som teller opp fra 1 til 100. Dette kan vi gjøre med en for-løkke, som i Python skrives slik:

```
for tall in range(1, 101):
    print(tall)
```

```
1
2
3
...
99
100
```

Her er utfordringen å huske hvordan man lagde en tallrekke, og akkurat hvordan grensene skulle være. Den enkleste måten å huske dette på er rett og slett å prøve seg frem. Kanskje du skriver koden, prøver å kjøre og oppdager at koden slutter på 99, istedenfor 100, da kan du enkelt gå tilbake og endre koden.

I neste steg kan vi fokusere på å erstatte alle tall delelige med 3, med "Fizz". Dette betyr at vi først må skjønne hvordan vi kan klare å sjekke om et tall er delelig med 3.

For å sjekke om et tall er delelig med et annet bruker vi en matematisk operasjon som heter *modulo*. Denne dekkes ofte ikke i ungdomsskolematematikken, men den er ganske enkel. Vi skriver modulo med prostenttegn, for eksempel `5 % 3`. Kort fortalt gir den resten ved en divisjon:

- 7 delt på 2 gir en rest på 1, fordi $7 = 3 \cdot 2 + 1$. Derfor blir

$$7 \ \% \ 2 = 1$$

.

- 6 delt på 3 gir ingen rest, fordi $6 = 2 \cdot 3 + 0$ Derfor blir

$$6 \ \% \ 3 = 0.$$

- 11 delt på 4 gir en rest på 3, fordi $11 = 2 \cdot 4 + 3$, Derfor blir

$$11 \ \% \ 4 = 3.$$

Når vi sier at et tall er delelig med et annet, så mener vi at resten er 0. Vi kan derfor kombinere en modulo utregning med en if-test, for å sjekke om hvert tall i rekka vår er delelig med 3.

```
for tall in range(1, 101):
    if tall % 3 == 0:
        print('Fizz')
    else:
        print(tall)
```

Nå bør vi kjøre programmet og sjekke at det oppfører seg riktig

```
1
2
Fizz
4
5
Fizz
7
...
```

Her er en vanlig fallgruve å gleme å bruke to likhetstegn (`==`), som fører til en kryptisk feilmelding. En annen fallgruve er å glemme å bruke en hvis-ellers setning, altså å bruke `else`. Her kan man f.eks få et program som skriver "1, 2, 3, Fizz, 4, ...", altså at den skriver "Fizz" i tilegg til, og ikke *istedenfor* tallet 3.

Når vi ser at alt virker som det skal kan vi ta et steg til. Vi kan nå legge inn at dersom tallet er delelig på 5, så skriver vi "Buzz". Vi kan jo begynne med å gjøre dette helt likt, ved å teste. Siden vi nå får tre mulige utfall i testen vår må vi bruker `elif`, som står for "else if":

```
for tall in range(1, 101):
    # Er tallet delelig på 3?
    if tall % 3 == 0:
        print('Fizz')

    # Eller er tallet delelig på 5?
    elif tall % 5 == 0:
        print('Buzz')

    # Hvis ingen av delene
    else:
        print(tall)
```

```
1
2
Fizz
4
Buzz
Fizz
7
...
```

Fra utskriften ser vi at det ser ut til å fungere bra for både 3-gangen og 5-gangen. Det er nå lett og tro at vi er ferdig, men det er en liten justering vi trenger. Husk at dersom et tall er delelig med *både* 3 og 5, så skal det skrives ut begge deler, altså "FizzBuzz". programmet vi har skrevet vil kun skrive ut "Fizz". Dette er slik en if-elif-else fungerer i Python, kun én av tilfellene vil treffe inn. Om vi hadde skrevet koden litt annerledes hadde det kanskje vært sånn at både "Fizz" og "Buzz" hadde blitt skrevet ut, men på hver sin linje, som også hadde vært feil.

For å fikse dette legger vi til nok én test, en som sjekker om tallet er delelig med begge. Dette testen må legges før de andre, prøv gjerne å forstå hvorfor. Med dette lagt inn blir hele programmet vårt

```
for tall in range(1, 101):
    # Er tallet delelig med både 3 og 5?
    if tall % 3 == 0 and tall % 5 == 0:
        print('FizzBuzz')

    # Eller er det delelig med bare 3?
```

```
    elif tall % 3 == 0:
        print('Fizz')

    # Eller er det delelig med bare 5?
    elif tall % 5 == 0:
        print('Buzz')

    # Hvis ingen av mulighetene over
    else:
        print(tall)
```

En annen løsning ville her vært å isteden først sjekke om tallet var delelig på 15, fordi fordi $3 \times 5 = 15$. Dette vil gjort selve programmeringen litt lettere, for vi hadde unngått bruken av `and`, men det krever en litt dypere matematisk forståelse.

Med dette har vi løst hele oppgaven. Vi har lagd et program som leker FizzBuzz helt perfekt. For mange vil dette vært utfordrende nok, men de mest ivrige ønsker kanskje å gå enda litt lengre. Utvidelser kan nå f.eks være å legge inn at programmet spør brukeren hvor høyt de skal telle, istedenfor å telle til 100 hver gang. Eller så kan man endre på reglene, hva med å sjekke etter tall som er delelig med 2 og 7, istedenfor 3 og 5? Eller man kan legge til andre regler, og her setter kreativiteten grensene. Kanskje vi for eksempel skal gå opp til 100, og så ned igjen? Eller hva med å si at alle tall i ti-gangen skal hoppes helt over?

6.3 Hvorfor FizzBuzz er en fin programmeringsoppgave

FizzBuzz er blitt en populær programmeringsoppgave fordi selve oppgaven er så enkel å forstå, men for å programmere den opp må man beherske en god del programmeringsforståelse. Her må man klare å kombinere en løkke med flere tester, og man må klare å forstå rekkefølgen og flyten i koden.

Det er også en fin oppgave fordi man kan programmere den opp stegvis, og ved hvert steg kan man sjekke om vi er på rett vei eller ikke. Dette gjør at oppgaven, selv om den kan være ordentlig utfordrende for nybegynnere, samtidig kan være overkommelig med nok innsats.

6.4 En alternativ fremgangsmåte

Vi vil også presisere at løsningen vi her skisserte kun er én mulig løsning på problemet, og det er fullt mulig å gå frem på andre måter. Løsningen vi skisserte er nok den vi føler er mest intuitiv for nybegynnere, men det er fullt mulig at man tenker ulikt og finner andre veier til mål. Dette er veldig typisk for programmering.

Under viser vi en kode som går frem på en litt annen måte for å løse *FizzBuzz*, denne beskriver vi ikke i detalj, men lar det stå som en utfordring til deg å prøve å forstå hvordan dette fungerer.

```
for tall in range(1, 101):
    # Lag tom tekst
    utskrift = ''

    # Er tallet delelig med 3?
    if tall % 3 == 0:
        utskrift += 'Fizz'

    # Er tallet delelig med 5?
    if tall % 5 == 0:
        utskrift += 'Buzz'

    # Er utskriften fortsatt tom?
    if utskrift == '':
        utskrift = tall

    print(utskrift)
```

Denne koden bruker en annen fremgangsmåte og tankegang en den første vi skisserte. Her kan man også diskutere hvilken av løsningene som er "best", men på det finnes det ikke noe fasitsvar. Faktumet er at begge kodene svarer på oppgaven, og begge er like riktige, selv om de går frem ulikt.

7 Funksjoner

Vi skal nå dekke et nytt grunnkonsept, nemlig *funksjoner*. Vi har så langt allerede sett, og brukt, funksjoner i Python. For eksempel er `print`, `input`, og `sqrt` alle eksempler på funksjoner som vi bruker.

Felles for disse funksjonene er at vi bruker dem for å utføre gitte oppgaver. Når vi bruker en funksjoner sier vi at vi *kaller* på funksjonen. For eksempel *kaller* vi på `print`-funksjonen for å skrive ut en beskjed, vi *kaller* på `input`-funksjonen for å stille brukeren et spørsmål, og vi kaller på `sqrt`-funksjonen for å regne ut kvadratroten av et tall.

Når vi kaller på en funksjon i Python bruker vi alltid vanlige, runde, parenteser, `()`, etter funksjonsnavnet. Ofte skriver vi noe innenfor disse parentesene, og dette er input vi sender inn i funksjonen, som funksjonen skal gjøre noe med. For eksempel, hvis vi skal regne ut $\sqrt{81}$ skriver vi `sqrt(4)`. Vi sier da at `4` er input til kvadratrotfunksjonen, eller *argumentet*.

7.1 Definere egne funksjoner

I tilegg til å bruke de innebyge funksjonene i Python, kan vi lage våre egne. Dette kaller vi å *definere* en funksjon. Fordelen med å lage egne funksjoner er at vi da kan gjenbruke dem så mye vi måtte ønske. De kan også gjøre programmer langt mer ryddige og enklere å forstå, jo større koder man skriver, jo viktigere er funksjoner.

Siden vi sier at vi *definerer* en funksjon bruker vi nøkkelordet `def` for å definere en funksjon i Python, dette er kort for "define", eller "definer" på norsk.

La oss se på et eksempel:

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

Her skriver vi først `def`, fordi vi ønsker å definere en funksjon, deretter skriver vi navnet vi ønsker for funksjonen vår, i dette tilfellet velger vi `f`. Deretter skriver vi parenteser og skriver inn hva slags *input* funksjonen ønsker. I dette tilfellet ønsker vi kun ett argument til funksjonen, og velger å kalle denne for x. Til slutt trenger vi et kolon, slik som i løkker og tester. Merk at `f` og `x` er valgfrie navn vi har valgt helt selv.

Etter denne linjen kommer innholdet i selve funksjonen, og her trenger vi et innrykk. Alle kodelinjer med innrykk etter definisjonslinja hører til inne i funksjonen. Det er disse kodelinjene som utføres når funksjonen kalles. Det som i dette tilfellet skjer er at vi regner ut en tallverdi, basert på den ukjente variabelen x, og så *returnerer* vi denne.

Å *returnere* betyr å sende tilbake ut av funksjonen. Vi kan altså tenke på det som returneres som *resultatet* av funksjonskallet.

Som du kanskje har forstått er funksjonen som vi her har definert, en matematisk annengradsligning. Vi kunne like gjerne ha skrevet den slik:

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

Når vi kjører kodesnutten over, så *definerer* vi funksjonen, det betyr at etter disse kodelinjene er kjørt, så eksisterer `f` som en funksjon vi kan bruke fritt senere i programmet vårt. Men det skjer ingenting mer når vi definerer funksjonen. Dersom vi ønsker å faktisk bruke funksjonen, eller å få noe *resultat* ut av den, må vi *kalle* på den.

Vi kan her for eksempel kalle på funksjonen ved å skrive `f(4)`, og da kjøres kodelinjene inne i funksjonen med $x = 4$, la oss teste dette selv:

```
1 print(f(4))
```

```
25
```

Om vi da regner ut for hånd for å sammenligne får vi

$$f(4) = 4^2 + 2 \cdot 4 + 1 = 16 + 8 + 1 = 25,$$

som er det vi forventet å få.

Fordelen med å definere dette som en funksjon er som nevnt at det gjør det utrolig lett for oss å *gjenbruke* formelen vår. Vi kan rett og slett kalle på funksjonen på nytt og på nytt, men forskjellig input hver gang

```
1 print(f(-10))
2 print(f(-5))
3 print(f(0))
4 print(f(5))
5 print(f(10))
```

```
81
16
1
36
121
```

Vi kan også lagre resultatet fra et funksjonskall til en variabel, istedenfor å skrive det rett ut

```
resultat = f(2)
```

Når vi skriver denne koden, så definerer vi en ny variabel, og det er returverdien fra funksjonen, som lagres i variabelen vi definerer.

7.2 Analogier for funksjoner

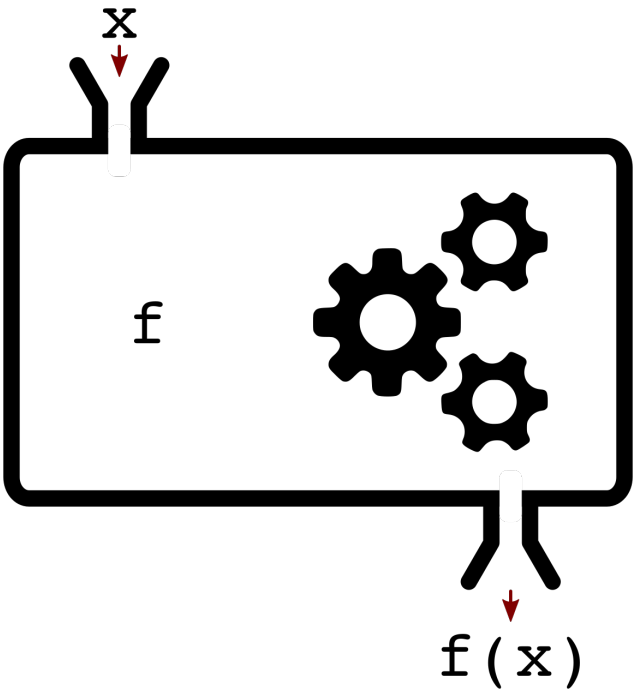
Funksjoner kan være et litt vanskelig konsept å forstå, så det er viktig å bruke tid og reflektere hva funksjoner egentlig *er*. For noen elever er funksjoner velkjente fra matematikken, og isåfall handler det mest om å generalisere konseptet til programmering. For andre er kanskje funksjoner i matematikken noe vanskelig og forvirrende, og isåfall kan det kanskje hjelpe og snakke om og bruker funksjoner i programmeringssammenheng. Uansett grunn kan det være nyttig å ha noen analogier for hva funksjoner er og gjør.

Kanskje den enkleste forklaringen på en funksjon er at det er en *regel*. Når vi definerer en funksjon, så lager vi en regel for hva som skje når vi bruker den. Her kan det for eksempel være nyttig å implementere funksjoner frakoblet fra datamaskin, og vi kommer til å dekke dette i neste seksjon.

En noe mer komplisert analogi som ofte brukes, er å tenke på en funksjon som en slags *maskin*. Denne maskinen har en åpning på den ene side, hvor vi kan mate noe inn. Når vi putter noe inn i maskinen, for eksempel et tall, begynner den å jobbe. Etter maskinen har blitt ferdig å jobbe, spytter den så ut resultatet på den andre siden. Denne maskinen er funksjonen vår. Selve maskinen har et navn, i dette tilfellet `f`. Vi kan nå mate inn noe i maskinen, her markert som en `x`, og vi får resultatet ut.

I programmeringsverden kaller vi gjerne en slik maskin for en "black box". Med det mener vi at selve innholdet i maskineriet er skjult og bygget inn. Det betyr at når man bruker en funksjon, så trenger vi ikke å tenke på hva innholdet i selve funksjonen/maskinen er, vi trenger bare å tenke på hva den egentlig gjør med inputen.

Tenk for eksempel på en brødbakemaskin. Instruksene sier hva du må putte inn i maskinen, og så får du et ferdig bakt brød ut. For å bruke denne maskinen trenger du ikke å vite hvordan den er skrudd sammen, eller hvordan man baker et brød.



De som har bygget brødbakemaskinen har tenkt på det. Det eneste vi trenger å skjønne for å få nytte av maskinen er hvilke ingredienser den tar inn, og hva vi får ut på den andre siden.

Det at funksjoner fungerer som svarte bokser, eller "black boxes" viser noe av styrken ved funksjoner. Ved å bare måtte tenke på de kompliserte detaljene når man lager funksjonen, og ikke når man bruker dem, så blir det en abstraksjon som hjelper oss bryte ned store og kompliserte problemer i enklere biter.

7.3 Frakoblede funksjoner

For å få en bedre forståelse for hva funksjoner er, og hvordan de fungerer, kan man jobbe med funksjoner frakoblet fra datamaskin, altså som en "unplugged" aktivitet. Her kan man for eksempel leke en enkel lek med to spillere, der man skal prøve å gjette hvilken funksjon den andre spilleren har implementert. Dette illustrerer også konseptet "black box".

Sett elever sammen i par. En av spillerene får rollen som funksjon, denne eleven får så enten utdelt en funksjon fra læreren, eller får lov til å skrive sin egen funksjon.

Den andre spilleren skal nå gi input til funksjonen. For hver input som gis må spilleren som har funksjonsrollen gå igjennom stegene i funksjonen og gi en verdi i retur.

Etter å ha gitt litt forskjellig verdier som input, og fått litt forskjellig verdier i retur, kan spilleren nå begynne å gjette seg frem til hva funksjonen er. Etter hver input-output som gis, får spilleren lov til å gjette på en funksjonsdefinisjon. Her bør man ikke gjette blindt, da blir det helt umulig å treffe. Isteden bør de forklare hvorfor de gjetter det de gjør, basert på inputen og outputen de har fått. Fortsett å gi input og output til spilleren klarer å gjette funksjonen, i hvilket tilfelle runden er over og de to spillerene bytter roller. For å gjøre litt konkurranse ut av det kan man si at det er om å gjøre å gjette riktig på så få input/output som mulig.

Før man leker denne leken bør man ha satt noen klare rammer på hva slags funksjoner man kan lage, og vanskelighetsnivået. Alternativ kan læreren lage funksjonskort på forhånd, som deles ut tilfeldig til elevene, dette er kanskje en god idé for å få opplegget til å gå som knirkefritt som mulig.

Eksempler på enkle funksjoner:

```
def f(x):
    return x + 4
```

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

```
def f(x):
    return x - 10
```

Eksempler på vanskeligere funksjoner:

```
def f(x):
    return 2*(x - 1)
```

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

Etterhvert som elevene lærer mer, kan man begynne å lage funksjoner som bruker mer av det man kan om Python. For eksempel en funksjon som tar inn to ting

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

Eller kanskje en løkke som også inneholder en if-test:

```
def f(x, y):
    if x > y:
        return x
    else:
        return y
```

Slike funksjoner kan fort bli såpass kompliserte at de blir vanskelig å gjette seg frem til, så her er enten gode retningslinjer, eller ferdiglagde funksjonskort gode valg.

7.4 Flere input, og flere output

Det er fullt mulig å lage en funksjon som tar mer enn én input. La oss for eksempel si vi ønsker å lage en funksjon som bruker Pytagoras' regel for å regne ut en hypotenus, da kan vi definere den som følger:

```
from pylab import *

def hypotenus(a, b):
    return sqrt(a**2 + b**2)
```

Merk at vi må importere `sqrt` for at funksjonen skal fungere, men dette trenger vi ikke nødvendigvis å gjøre inne i funksjonen, det holder å gjøre det i toppen av programmet vårt.

Tilsvarende kan vi fint lage en funksjon som returnerer mer enn én ting, da legger vi dem bare etterhverandre og deler dem med komma.

```
def divisjon(teller, nevner):
    heltall = teller//nevner
    rest = teller % nevner
    return heltall, rest


a = 9
b = 5
htall, rest = divisjon(9, 5)
print("{}/{} = {} + {}/{}".format(a, b, htall, rest, b))
```

```
9/5 = 1 + 4/5
```

7.5 Funksjoner og Variabler

Du har nå lært hvordan du kan opprette eller definere variabler, og hvordan du kan definere funksjoner. Her kan det være verdt å reflektere at funksjoner i seg selv faktisk *er* variabler. Som andre variabler har de et navn, og et innhold. I dette tilfeller har derimot ikke variabelen tekst eller tall som innhold, men *kode*, og denne koden kjøres når man bruker funksjonen.

For å holde ting ryddig i språket, anbefaler vi derimot å ikke referere til funksjoner til elever som variabler, da det kan føre til litt forvirring. Istedet anbefaler vi heller å konsekvent å kalle dem for funksjoner, eventuelt "funksjonsvariabler", for å tydelig skille dem fra andre variabler, som tall og tekststrenger.

7.6 Funksjoner i programmering – mer enn bare matematikk

Eksemplene vi har sett på av egendefinerte funksjoner så langt har vært av matematisk natur. I matematikk er funksjoner alltid noe som regner på tall. I programmering derimot, kan funksjoner gjøre nesten hva som helst. Dette er fordi funksjoner kan inneholde hva som helst av kode og det betyr at de kan gjøre alt som vi ellers gjør i programmene våre.

For eksempel kan vi lage en funksjon som tar et navn inn som argument, og så skriver ut en beskjed til personen.

```
1 def hils_på(navn):
2     print(f"Hei på deg {navn}!")
```

Nå kan vi bruke funksjonen som ellers

```
hils_på("Pål")
hils_på("Kari")
```

```
Hei på deg Pål!
Hei på deg Kari!
```

Merk at funksjonen her inneholder `print`-kommandoen, og skriver derfor rett ut til skjermen når vi bruker den. Dette er ulikt fra våre tidligere eksempler, hvor vi isteden brukte `return` for å *returnere* en tallverdi, som vi så skrev ut utenfor funksjonen.

La oss ta et litt lengre eksempel. I den engelske bursdagssangen *"Happy Birthday to You"* bygger man navnet på bursdagsbarnet i sangen. La oss gjøre dette i en funksjon

```
def happy_birthday(navn):
    print('Happy birthday to you!')
    print('Happy birthday to you!')
    print(f'Happy birthday dear {navn}!')
    print('Happy birthday to you!')
```

Nå er vi klare for å feire bursdag, og vi kan enkelt kalle på funksjonen med et navn som argument:

```
happy_birthday("Anne-Mari")
```

```
Happy birthday to you!
Happy birthday to you!
Happy birthday dear Anne-Mari!
Happy birthday to you!
```

8 Sammensatt eksempel 2: Primtallssjekker

La oss se på et nytt eksempel som krever litt mer sammensatt programmering, nemlig å lage et program som sjekker om et gitt tall er et primtall eller ikke.

Denne oppgaven er fin fordi den dekker mye av det som er dekket i dette kurset, og det passer også godt inn i matematikkpensum. Derimot er den litt mer utfordrende, og kan kreve at man holder elevene litt mer i hånda gjennom opplegget.

8.1 Frakoblet

Før man setter igang å prøver å skrive kode som finner primtall, så er det lurt å ta et steg tilbake og repetere hva primtall egentlig er, og hvordan vi kan sjekke om

et tall er et primtall eller ikke for hånd.

Definisjonen på et primtall er et positivt heltall som kun kan deles på 1 eller seg selv. Litt rart er derimot at tallet 1 faktisk ikke er et primtall.

Nå kan det være lurt å stille elevene et spørsmål, og la dem tenke igjennom eller diskutere det med sideeleven:

- Hvis jeg gir deg et tall, for eksempel 13, hvordan sjekker du om det er et primtall?

Når de har fått litt tid til å tenke eller diskutere gjennom dette, spør dem om de klarer å lage en algoritme/oppskrift som kan brukes for et *hvilket som helst tall*.

For eksempelet med 13 kan vi sjekke om det er primtall ved å sjekke om 13 er delelig med et tall i rekka

$$2, 3, 4, \ldots, 12.$$

For hvert tall i rekka må vi prøve å dele:

$$\begin{aligned}
13/2 &= 6.5 \\
13/3 &= 4.33.... \\
13/4 &= 3.25 \\
13/5 &= 2.6 \\
13/6 &= 2.166... \\
13/7 &= 1.857... \\
13/8 &= 1.625 \\
13/9 &= 1.444... \\
13/10 &= 1.3 \\
13/11 &= 1.181... \\
13/12 &= 1.083...
\end{aligned}$$

Så vi ser at 13 er ikke delelig med noen av tallene, derfor *er* 13 et primtall.

Om vi prøver med et annet tall, f.eks, 15, finner vi:

$$\begin{aligned}
15/2 &= 7.5 \\
15/3 &= 5
\end{aligned}$$

Her ser vi allerede på vårt andre tall at 15 er delelig med 3, derfor er 15 *ikke* et primtall, og vi slipper å prøve resten av tallene opp til 15.

Her har vi allerede snublet over en generell oppskrifteller algoritme for problemet. Denne algoritmen er kanskje allerede vandt til å gjennomføre for hånd, ihvertfall hvis de har lært om primtall i matematikken. Derimot blir den fort kjedelig å gjøre for store tall! Er f.eks 11203 et primtall? Om vi klarer å oversette kunnskapen vi har til datamaskinkode så har vi et program som kan sjekke dette for oss lynkjapt.

8.2 Å programmere en primtallssjekker

Vi skal nå begynne å skrive en program som kan sjekke om et tall er et primtall eller ikke. Vi velger å gjøre dette som en funksjon, da det kommer til å gjøre et par ting litt lettere for oss.

Funksjonen vår velger vi å kalle `er_primtall`, og da begynner vi å definere den slik som dette:

```
1 def er_primtall(n):
2     ...
```

Her har vi valgt å kalle tallet vi skal sjekke for n, fordi dette er et vanlig navn for et heltall i matematikken. Om vi ønsker kan vi godt kalle den for noe annet isteden, f.eks `tall`.

Nå må vi fylle inn for innholdet i funksjonen, altså der det står skrevet `...`.

Det vi må gjøre i funksjonen vår er å gå igjennom tallene

$$2, 3, \ldots n-1.$$

Dette kan vi få til med en `for`-løkke. Så vi kan skrive inn dette i funksjonen vår:

```
1 def er_primtall(n):
2     for d in range(2, n):
3         print(d)
```

Her bruker vi en `for`-løkke for å løkke over tallrekka fra 2 opp til n. Vi velger å kalle løkkevariabelen for d, men dette er igjen valgfritt. Om du syns der logisk å kalle den noe annet, f.eks `divisor` eller `mulig_divisor`, så gjør gjerne det

For å sjekke om vi er på rett spor bør vi stoppe opp og teste koden vår, derfor har jeg valgt å skrive inn at vi skal skrive ut `d` med en `print`-kommando. Denne kommer jeg til å fjerne igjen når jeg ser at ting fungerer, men den er fin å ha akkurat nå.

Nå sjekker vi programmet ved å kalle på funksjonen med et eksempel, f.eks 13, sånn som istad

```
er_primtall(13)
```

```
2
3
4
5
6
7
8
9
10
11
12
```

Vi ser nå at programmer går igjennom de tallene vi ønsker, nemlig fra og med 2, opp til, men ikke med, tallet n. Her er det lett å ha gjort feil i `range`-kommandoen, som man isåfall må fikse opp før man fortsetter.

Når vi vet at løkka er rett, så kan vi bytte ut `print`-kommandoen med det vi egentlig vil gjøre inne i løkka, nemlig sjekke om n er delelig med d. Dette gjør vi på samme måte som vi forklarte i *FizzBuzz* nemlig ved å sjekke resten i divisjonen med modulooperatoren. Vi skriver derfor en `if`-test som sjekker dette. Hvis if-testen er sann, altså, hvis n er delelig med d, så vet vi at n *ikke* er et primtall. Derfor skriver vi `return False`, sånn som dette:

```
def er_primtall(n):
    for d in range(2, n):
        if n % d == 0:
            return False
```

Her betyr `False` at vi svarer på spørsmålet "er primtall" med *nei*. Så fort vi finner et tall som deler n, så *returnerer* funksjonen, og vi sjekker ingen flere tall.

La oss sjekke et tall vi *vet* ikke er et primtall, f.eks 10:

```
print(er_primtall(10))
```

```
False
```

Svaret "False" betyr altså *falskt*, altså er 10 ikke et primtall. Vi skal forbedre på dette svaret etterhvert.

Hva om vi prøver et tall vi vet er primtall, f.eks 7?

```
print(er_primtall(7))
```

```
None
```

Her *burde* svaret vært "True", fordi 7 er et primtall. Derimot får vi "None", som er det vi får når en funksjon ikke har returnert noe. Her returnerer funksjonen vår aldri noe, fordi if-testen er aldri sann. Vi kan løse dette ved å få funksjonen til å returnere sant dersom hele løkka fullfører uten å finne noe som deler tallet vårt:

```
def er_primtall(n):
    for d in range(2, n):
        if n % d == 0:
            return False

    return True
```

Nå begynner koden vår å bli ordentlig kompleks her, og det er virkelig verdt å ta et steg tilbake og virkelig tenke igjennom hva som foregår her.

Merk at linja vi la til, altså `return True` er helt i bunn, og hvis du ser litt på innrykkene hører den *ikke* til løkka, den kjøres altså kun etter hele løkka er ferdig. Derfor returnerer vi altså True *kun* hvis vi ikke finner ett eneste tall i rekka som deler n.

Vi kan nå prøve koden vår igjen, med et par tall

```
print("2 er primtall?", er_primtall(2))
print("3 er primtall?", er_primtall(3))
print("4 er primtall?", er_primtall(4))
print("5 er primtall?", er_primtall(5))
```

```
2 er primtall? True
3 er primtall? True
4 er primtall? False
5 er primtall? True
```

Her ser vi at funksjonen ser ut til å fungere som vi ønsker. Derimot er det én liten ting som blir feil. Om man ikke tenker seg nøye om kan man tro man er helt ferdig nå, men tallet 1 vil gi feil svar. Prøv selv, om du sjekker vil funksjonen si at 1 *er* et primtall. Dette er et godt sted å stoppe opp og spørre seg selv *hvorfor skjer dette?*

Svaret er rett og slett at tallrekka `range(2, n)` skal gå fra 2 til, men ikke med, n. Så om $n = 1$, så vil løkka aldri kjøre, og vi går rett til `return True`. Å fikse dette er egentlig ikke så lett, fordi 1 er et litt spesielt tilfelle, altså må vi nesten rett og slett legge til 1 som et spesialtilfelle. Med dette blir hele funksjonen:

```
1  def er_primtall(n):
2      # Spesialtilfelle: 1 er ikke et primtall
3      if n == 1:
4          return False
5
6      # Let etter divisor
7      for d in range(2, n):
8          if n % d == 0:
9              # En divisor er funnet, n er derfor ikke
                   primtall
10             return False
11
12     # Prøvd alle tall uten å finne en divisor
13     return True
```

Nå har vi laget en funksjon som kan sjekke om et hvilket som helst positivt heltall n er et primtall. Vi kan egentlig stoppe her, eller vi kan gå et skritt lenger og legge på litt input/output funksjonalitet:

```
1  brukerens_tall = input("Hvilket tall skal jeg sjekke? ")
2
3  if er_primtall(brukerens_tall):
4      print(f"{brukerens_tall} er et primtall!")
5  else:
6      print(f"{brukerens_tall} er ikke et primtall")
```

Når vi nå kjører programmet vårt kan vi fylle inn et tall vi ønsker at programmet vårt skal sjekke for oss

```
Hvilket tall skal jeg sjekke? 40193
40193 er et primtall!
```

En annen mulighet er å kombinere funksjonen vår med en for-løkke for å finne alle primtall under en viss grense. Vi kan f.eks finne alle primtall under hundre:

```
1 for n in range(1, 100):
2     if er_primtall(n):
3         print(n)
```

```
2
3
5
7
11
13
17
...
79
83
89
97
```

8.3 Læringsutbytte

Oppgaven med å lage en primtallssjekker kan være en stor utfordring for en nybegynner i programmering, og om man skal gjøre dette som et opplegg med elever er det viktig å legge det opp på en måte der elevene veiledes steg for steg om de setter seg fast.

Det krever altså litt innsats fra dere som lærere å arbeide med dette stoffet. Vi vil derimot argumentere for at det potensielt foreligger et stort læringsutbytte for elevene.

For det første er dette et godt eksempel på en god programmeringsoppgave. Selv om selve sluttresultatet ikke trenger voldsomme mengder kode, så krever løsningen faktisk alle de grunnleggende elementene vi skal dekke i programmeringen.

Variabler, løkker, tester, og funksjoner er alle viktige elementer. Samtidig krever oppgaven en god dose algoritmisk tankegang, der man kan bryte ned et forholdvis komplekst problem til mindre biter og gå frem stegvis for å løse dem.

I tilegg til å være en god programmeringsoppgave kan det være en god matematikkoppgave. I prosessen av å jobbe med koden må man tenke nøye igjennom og reflektere rundt ulike antagelser man gjør og hvilke slutninger man kan dra. For noen elever kan det vært utrolig nyttig å se stoffet på en ny måte, og på den måten repetere og forsterke konsepter.

For at en skal få nytte av opplegget er det viktig at elevene tvinges til å reflektere og diskutere rundt oppgaven. Her er det lurt å få elever til å jobbe i par, og legge inn konkrete diskusjonsoppgaver, der de må fortelle hverandre hva de tenker. På den måten danner oppgaven et grunnlag for diskusjon.

9 Plotte tallrekker

9.1 Lister

Nå skal vi straks lære å bruke Python for å lage fine kurveplot av samlinger av tall. Men før vi gjør det må vi lære hvordan vi kan lagre en samling av tall i en variabel. Hittil har du sett at variable har et navn, et innhold og en type. La oss se på en ny type variabel: lister. La oss si at du ikke bare ønsker at programmet ditt skal huske på et navn, men en hel skoleklasse. Det er veldig slitsomt å måtte opprette en variabel for hver enkelt elev. Det vi kan gjøre, er å opprette en enkelt variabel, hvor vi lagrer alle elevene sammen, det kan du gjøre sånn her

```
1 elever = ["Alma", "Filip", "Sofia", "Mohammad", "Alex"]
```

Vi ser at vi har brukt firkantparanteser, `[` og `]`, for å definere en liste (disse parantesene kalles også gjerne for *klammeparanteser*. Inne i listen har vi skrevet 5 navn, alle adskilt med komma. Merk også at vi definerer hvert navn som hver sin tekststreng. Når du har definert en liste på denne måten, så kan du skrive den ut med `print`.

```
1 print(students)
```

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

```
['Alma', 'Filip', 'Sofia', 'Mohammad', 'Alex']
```

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

```
print(len(elever))
```

```
5
```

forteller oss at det er fem navn i lista.

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

```
prisliste = [299, 199, 4000, 20]
```

Eller en blanding av tall og tekst

```
godt_og_blandet = ["litt tekst", 2, 2.3, 9, "litt mer
   tekst"]
```

Du kan til og med legge lister inne i andre lister

```
liste_av_lister = [[4, 10, 12], ["Tom", "David", "Peter"]]
```

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

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

```
elever = ["Alma", "Filip", "Sofia", "Mohammad", "Alex"]
```

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

```
print(elever[0])
print(elever[3])
```

```
Alma
Mohammad
```

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

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

```
students[4] = "Alexander"
```

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

```
['Alma', 'Filip', 'Sofia', 'Mohammad', 'Alexander']
```

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

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

```
students.append("Sara")
```

Hvis vi nå skriver ut lista får vi

```
['Alma', 'Filip', 'Sofia', 'Mohammad', 'Alexander', 'Sara
   ']
```

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

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

```
min_liste = []
min_liste.append(1)
min_liste.append(2)
min_liste.append(3)
print(min_liste)
```

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

Notis til klasserommet

I matematikken har vi som oftest kun bruk for lister av tall. Det kan i sånne sammenhenger virke litt forvirrende og vise frem alle de forskjellige tingene man faktisk kan putte i en liste. Her er det derfor lurt å tenke igjennom hva det er man vil bruke lister til. Hvis man kun skal bruke lister for matematiske oppgaver og plotting, holder det kanskje å se på lister av tall. Det er uansett nyttig for dere å være klar over at lister kan inneholde hva som helst.

9.2 Lister for å lagre resultat fra løkker

Det hender ofte at vi ønsker å lagre resultatet fra en liste og ikke bare printe det ut. I seksjonen om `while`-løkker lagde vi et program for å undersøke utviklingen av sparepengene på BSU konto:

```
penger = 10000
år = 0
rente = 1.035

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

print(f'Du må vente i {år} år.')
print(f'Det er da {penger} på konto.')
```

```
Du må vente i 21 år.
Det er da 20594 kr på konto.
```

Dette programmet tar kun vare på hvor mange penger du har til slutt. Resten blir skrevet over. Hva om du har lyst til å lagre mengden penger du har for hvert år? Du kan ikke bruke variabler, for da trenger du en variabel for hvert år og siden du bruker en `while`-løkke vet du ikke hvor mange år det er snakk om. Da er lister nyttige å ha. Det du kan gjøre er å opprette en tom liste før løkka. Så kan du i hver iterasjon utvide lista med `append`. Dette gjør at du til slutt får en liste som inneholder et tall for hver "runde" i løkka. Programmet blir da seendes slik ut:

```
penger = 10000
år = 0
```

```
rente = 1.035

penge_liste = []
while penger < 20000:
    penger *= rente
    år += 1
    penge_liste.append(penger)

print(f'Du må vente i {år} år.')
print(f'Det er da {penger} på konto.')
print('Pengeutvikling:')
print(penge_liste)
```

```
Du må vente i 21 år.
Det er da 20594 kr på konto.
Pengeutvikling:
[10350.0, 10712.25, 11087.17875, 11475.230006249998,
   11876.863056468746, 12292.553263445152,
   12722.79262766573, 13168.09036963403,
   13628.97353257122, 14105.987606211213,
   14599.697172428603, 15110.686573463603,
   15639.560603534828, 16186.945224658546,
   16753.488307521595, 17339.86039828485,
   17946.75551222482, 18574.891955152685,
   19225.01317358303, 19897.888634658433,
   20594.314736871478]
```

Denne gangen får vi tallverdiene for hvert år samlet i en liste.

Nå lurer du kanskje på hvorfor vi ønsker å ha pengene i en slik liste? Når vi skriver den ut er det jo ikke spesielt pent, og det er ikke så lett og se på utviklingen som en haug med tall? Det er sant at lister med tall ikke alltid er så interessante å se på når vi skriver dem ut. Men dersom vi har lagret resultatene som en liste med tall kan vi også plotte dem. Dette gjør at vi visualisere hvordan pengemengden øker mer og mer for hvert år. Plotting skal vi lære om i neste seksjon.

9.3 Grafer

Vi skal nå se på hvordan vi kan bruke Python til å plotte data og tegne grafer. Det å lage grafer er et ganske stort tema, og vi kommer bare til å dekke de enkleste formene for plotting i dette kompendiet—men dette er fortsatt et viktig verktøy, det er det grunnleggende som er viktigst. Dette er i mange tilfeller nok til å illustrere funksjoner, resultatet og sammenhenger for å virkelig skjønne hva som foregår.

For å plotte skal vi bruke et tilleggsbiblotek som heter Matplotlib. Dette kan importeres slik:

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

eller

```
import matplotlib.pyplot as plt
```

På denne måten har vi tilgang på alt vi trenger for å plotte i programmene våres. I dette kompendiet skal vi bruke den første måten å hente plottefunksjonaliteten.

Den viktigste typen plott vi skal se på i dette kurset er kurveplott, eller grafer. Når vi tegner en graf for hånd gjør som oftest dette ved å tegne et sett med (x, y)-punkter, og så trekker vi en linje igjennom disse punktene. Python gjør dette på tilsvarende måte. Når vi skal lage et kurveplott må vi derfor lage to sekvenser med data, der den ene er verdiene langs x-aksen, og den andre er verdiene langs y-aksen.

9.3.1 Eksempel

Én måte å huske en rekke verdier er å bygge opp en liste mens vi jobber. Om vi har definert en liste med firkantparenteser (`[]`) så kan vi legge til elementer til denne lista ved å bruke spesialfunksjonen `append`. Det er enklest å forklare ved å vise:

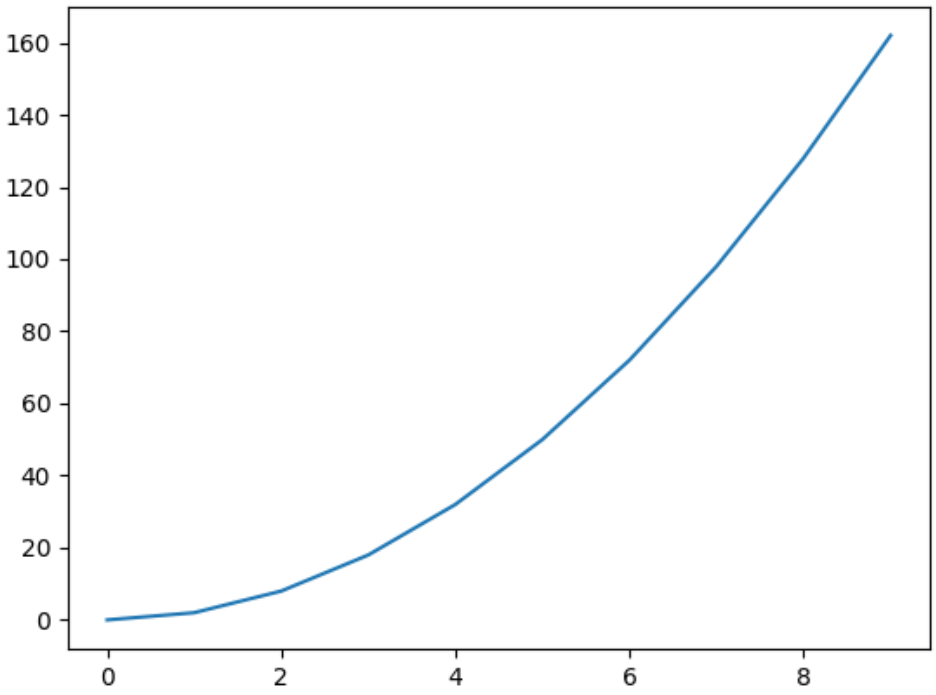
```
xvverdier = []
yverdier = []

for x in range(10):
    xverdier.append(x)
    yverdier.append(2*x**2)
```

I dette eksempelet fyller vi inn verdier for funksjonen $y = 2x^2$. Før loopen kjører inneholder listene ingen verdier. Men hver gang løkka repeterer seg legger vi til en ny x-verdi og en ny y-verdi. Før vi går videre kan du prøve å printe ut listene for å se hvordan de ser ut.

Nå som vi har laget de to listene våres kan vi lage et plot ved å kalle på `plot`-funksjonen. Denne tar x-verdiene som første argument, og y-verdiene som andre argument. Om vi ønsker tiden på x-aksen må vi derfor skrive `plot(t, s)`. For å faktisk få frem plottet må vi også kalle på `show()`. Dette blir litt som å huske å printe en variabel for å faktisk se den.

```
from matplotlib.pyplot import *

# Fyll inn kode for å finne t- og s-verdiene

plot(xverdier, yverdier)
show()
```

Figuren vi får ut ser ut som følger:



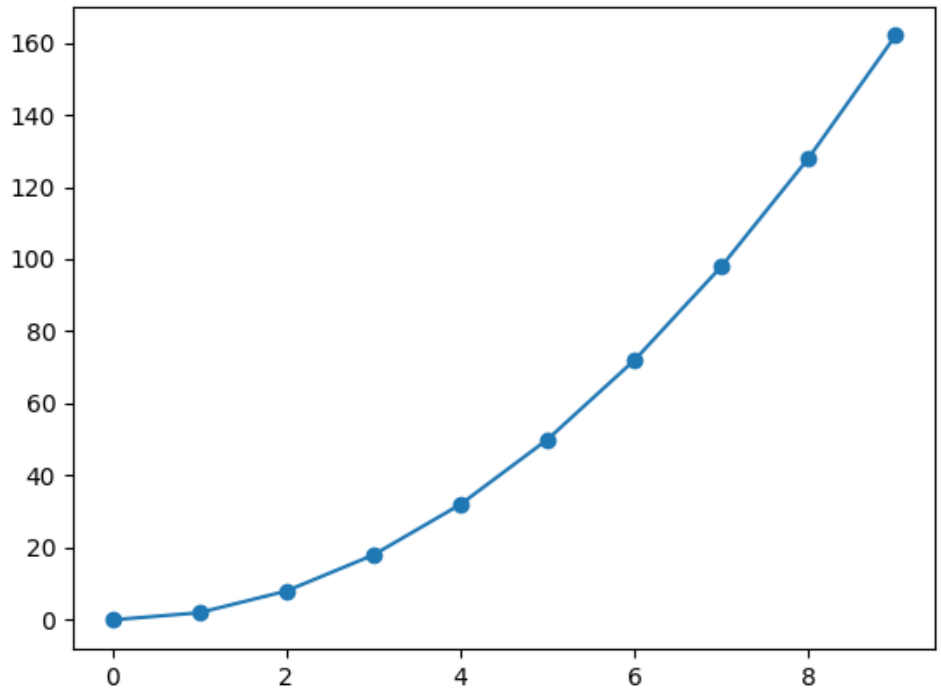
Figur 8: Funksjonen $y = 2x^2$ Denne figuren er det vi får når vi kun bruker `plot()`, uten noe ekstra "pynt".

/

Dette ser ut som en kurve, en linje med krumning, men er egentlig bare en rekke rette linjer som knytter sammen datapunktene vi fant i løkka vår. Vi kan vise dette tydeligere ved å legge til en liten `'o-'` til `plott`-argumentet vårt:

```
1 plot(t_verdier, s_verdier, 'o-')
```

Her betyr `'o-'` at man skal plotte sirkler på selve datapunktene, og streker imellom. Andre valg er `'o'` (bare datapunktene), `'x'` (kryss på datapunktene), `'--'` (stiplet linje), osv. Resultatet av å bruke `'o-'` er vist her:



Figur 9: Ballens høyde over bakken som funksjon over tid når vi eksplisitt viser frem hvert datapunkt.

9.4 Pynte på plott

Selve kurva over kommer rett fra formelen, og er derfor akkurat slik den bør være. Derimot bør vi også legge til navn på aksene våres, og pynte på plottet på et par andre måter. Dette gjør vi ved å skrive inn et par ekstra kodelinjer imellom `print()` og `show()` funksjonene våres.

```
plot(t_verdier, s_verdier)
xlabel('Tid [sekunder]')
ylabel('Høyde over bakken [meter]')
title('Vertikalt kast')
axis([0, 3, 0, 10])
show()
```

Her legger funksjonene `xlabel()` og `ylabel()` navn på aksene. Tilsvarende legger `title()` et navn på toppen av figuren. Til slutt har vi brukt `axis()` til eksplisitt å sette grensene på x- og y-aksen. Denne funksjonen tar en liste som argument av typen `[xmin, xmax, ymin, ymax]`. Så her lar vi figuren gå fra 0 til 3 sekunder, og fra 0 til 10 meter. Vi kunne også lagt på et rutenett med `grid()` (uten argument), men dette kan du prøve selv. Resultatet blir som følger:

9.4.1 Eksempel: Plotte andregradsfunksjoner

La oss ta et til eksempel. La oss si vi ønsker å plotte to andregradsfunksjoner:

$$f(x) = -x^2 + 4x + 3,$$
$$g(x) = x^2 + 3x - 7.$$

Om vi ønsker å plotte to kurver i samme figur må vi bruke to `plot()`-kommandoer etterhverandre.

Senere i kapitellet skal vi se hvordan vi kan plotte matematiske funksjoner av denne typen langt mer effektivt, men for nå gjentar vi samme fremgangsmåte som over.

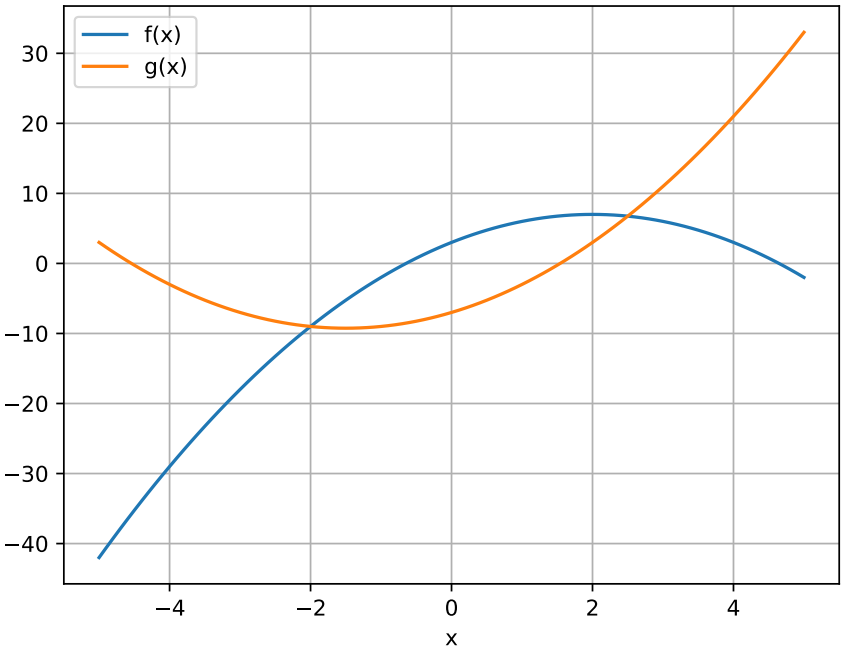
Før vi plotter må vi bestemme os for hvilke x-verdier vi ønsker, la oss si fra -5 til t. Deretter regner vi ut tilsvarende y-verdier:

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

xverdier = []
fverdier = []
gverdier = []

for x in arange(-5, 5.1, 0.1):
    f = -x**2 + 4*x + 3
    g = x**2 + 3*x - 7

    xverdier.append(x)
    fverdier.append(f)
    gverdier.append(g)

plot(x, f, label='f(x)')
plot(x, g, label='g(x)')
xlabel('x')
```

```
19 legend()
20 grid()
21 show()
```

Her har vi brukt same metode som tidliger. Det som har endret seg er at vi nå bruker to `plot()`-kall, en for hver farge. Vi bruker da tileggsargumentet `label='f(x)'`, dette er ikke strengt tatt nødvendig, men lar oss bruke `legend()` for å kunne skille på de to kurvene.

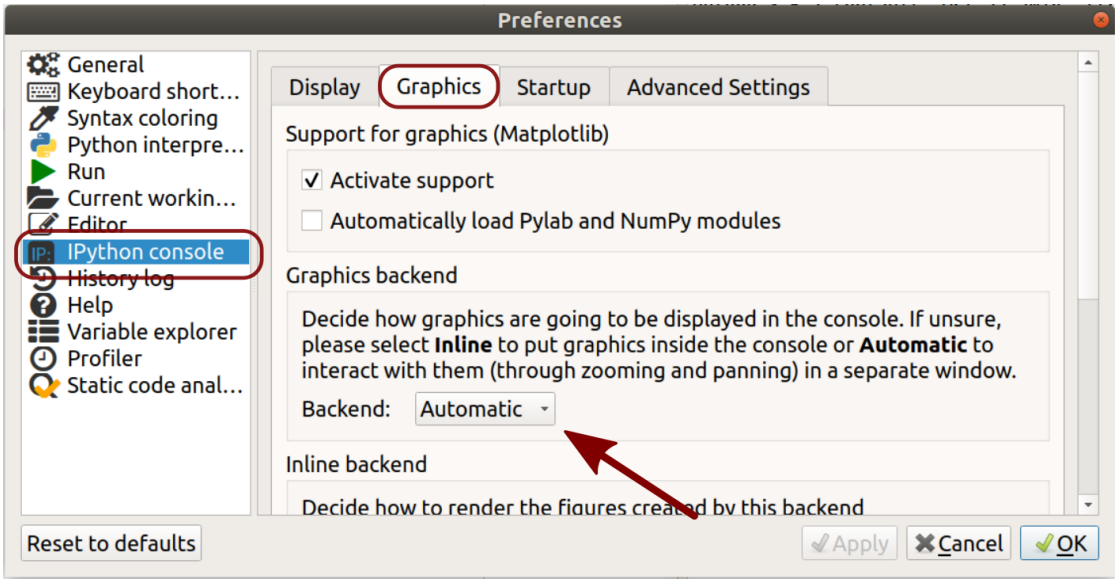
Resultatet blir som vist i Figur 10. Merk at de to kurvene automatisk får ulike farger, og fordi vi brukte `legend()` får vi en nøkkel som knytter de ulike fargene til de ulike kurvene.



Figur 10: To annengradsfunksjoner plottet i samme figur.

9.5 Grafikk i Sypder

Om du har prøvd det første plotte-eksempelet i Spyder ser du at grafikken dukker opp inne i konsollen, tilsvarende annen output. Dette er på en måte fint, ettersom



Figur 11: Det kan være en god idé å endre innstillinger for hvor figurer vises slik at de ikke havner inne i konsollen. Her kan man også huke av for å automatisk bruke pylab, om man ønsker å slippe å importere i hvert program man bruker.

at det er det man er vandt med. Derimot kan figurene ofte bli for små når de skal passe inn i konsollen, og vi får ikke sett dem skikkelig. I tilegg kan vi ikke interegere med plottene for å f.eks å zoome inn på stilige detaljer og lignende.

På grunn av dette anbefaler vi å endre instillingene for hvordan grafikk vises i Spyder. Om du går inn på følgende instillinger:

- `Tools` → `Preferences` → `IPython Console` → `Graphics`

Der kan du endre instillingen for `Graphics Backend` fra `Inline` til `Automatic`. Etter dette må du restarte Spyder for at endringene skal tre i kraft.

Etter at du har restartet Spyder kan du prøve å kjøre programmet ditt på nytt. Denne gangen skal forhåpentligvis figuren ikke dukke opp i konsollen, men som et eget vindu, her har du nå mulighet til å interagere dynamisk med plottet, og kan endre navn på akser, flytte på vinduet og lignende.

9.6 Lagre plott og grafer

Mens vi jobber med koden vår produserer vi gjerne det samme plottet gang på gang, med små variasjoner for hver gang. Når vi er fornøyd med resultatet ønsker vi kanskje å lagre bildet så vi for eksempel kan inkludere det i en rapport, eller på en presentasjon.

Det finnes et par ulike måter å gjøre dette på, vi vil forklare de to mest frem måtene å gjøre det på.

9.6.1 Lagre bilder ved hjelp av kode

Kanskje den aller beste måten å lagre bilder på, er å gjøre det i selve koden med kodelinja

```
1 savefig("ballkast.png")
```

Denne kommandoen bør vi legge rett før `show()` kommandoen. Når denne kodelinja kjøres vil det opprettes en bildefil på maskinen med navnet `ballkast.png`. Navnet kan selvfølgelig endres som vi ønsker. Filen vil legges i samme mappe som selve koden vår, så pass på hvor programmet ditt er lagret om du lagrer bilder på denne måten. Merk at matplotlib også kan lagre i andre formater som for eksempel `.jpg`, `.pdf` eller `.svg`. Fordelen med de to sistnevnte er at disse er såkalt *vektorgrafikk*, som skalerer utifra hvordan de brukes, og ser derfor gjerne bedre ut på projektor og print.

9.6.2 Lagre bilder "manuelt"

Den andre måten vi kan lagre bilder på er interaktivt igjennom selve figuren. Om du har endret på Spyder-innstillingene så figuren dukker opp i et eget vindu vil det være et eget lagre-ikon i verktøylinja, eller man kan bruke den velkjente `Ctrl+S` hurtigtasten. Her kan man selv velge hvor bildet lagres via et vanlig grafisk grensesnitt.

Om man foretrekker å plotte figurer rett i konsollen kan du høyreklikke på et bilde i konsollen og velge å kopiere eller lagre det. Derimot vil du nok erfare at disse bildene desverre blir litt lav oppløsning. Dette kan du isåfall endre ved å øke oppløsningen under de de samme innstillingene som diskutert over, se under

`Inline Backend` i bunn av Figur 11.

Med `savefig()` så vil bildet lagres på nytt hver gang koden kjøres, på denne måten vil bildet alltid oppdatere seg, selv om vi gjør justeringer til koden vår. Om vi bruker "manuell" lagring vil bildet *ikke* oppdatere seg selv, og det er derfor potensielt mer jobb om man skal oppdatere ting etterhvert. Erfarne kodere foretrekker derfor gjerne `savefig` over manuell lagring.

9.7 Vektoriserte beregninger

Vi har sett hvordan `plot`-funksjonen tar to sekvenser med verdier langs henholdsvis x- og y-aksen. I eksemplene vi har sett så langt har vi funnet disse ved hjelp av en løkke. Vi skal nå se hvordan vi kan plotte matematiske funksjoner med langt mindre kode, derimot er prosessen litt mindre gjennomsiktig.

9.7.1 Vektorer i Python

Vi har i dette kompendiet med vilje ikke gått i for mye detaljer på hvordan lister fungerer, av hensyn til tid. Kort fortalt er lister, slik vi har brukt dem, en sekvens av verdier eller objekter. Derimot er lister ikke ment å regnes med. Om vi har en liste av verdier kan vi ikke regne med denne direkte. Om du for eksempel prøver følgende kode:

```
# OBS: Koden vil ikke fungere og gir feilmelding
x = [-2, -1, 0, 1, 2]
y = x**2
```

Vil du få en feilmelding av typen

```
TypeError: unsupported operand type(s) for ** or pow(): '
   list' and 'int'
```

Denne meldingen ser kanskje litt kryptisk ut, men den forteller oss at Python ikke skjønner hva potensen av en liste er for noe.

Dette skjer fordi en liste ikke er ment å regnes med. Derimot har vi en annen type sekvens som vi kan regne med, disse kalles *arrays* i Python. Om vi konverter lista vår til array kan vi regne med den direkte:

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

x = array([-2, -1, 0, 1, 2])
y = x**2
print(y)
```

```
[4 1 0 1 4]
```

Arrays er altså tiltenkt matematiske utregninger, og egner seg derfor langt bedre enn lister. Når vi regner på et helt array av gangen, slik som i dette eksempelet kaller vi gjerne det for en *vektorisert beregning*, det er fordi `x`-variabelen nå vil tilsvare en matematisk vektor.

For å opprette array kan vi enten konvertere en liste vi har skrevet ut manuelt, slik som i eksempelet over, eller vi kan bruke to veldig nyttige funksjoner fra `numpy`: `arange` og `linspace`.

9.7.2 Bruk av `arange`

Vi har allerede sett `arange`, denne oppfører seg likt som `range`, men kan jobbe med desimaltall. En annen forskjell er at `arange` faktisk lager et array. Altså kunne vi forenklet plotting av andregradsfunksjonene våres til følgende kode:

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

x = arange(-5, 5.1, 0.1)
f = -x**2 + 4*x + 3
g = x**2 + 3*x - 7

plot(x, f)
plot(x, g)
show()
```

Ettersom at x er laget med `arange`, blir det en vektor, og dermed vil f og g automatisk bli det også. Dette er altså langt mer effektivt når vi skal regne ut en lang rekke verdier.

9.7.3 Bruk av linspace

Den andre funksjonen, `linspace`, har vi ikke sett enda. Det noe rare navnet står for: "linear spacing". Om vi skriver `linspace(a, b, n)` får vi n jevnt fordelte punkter på intervallet $[a, b]$. For eksempel:

```
x = linspace(-5, 5, 11)
print(x)
```

```
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.]
```

Her har vi altså 11 punkter jevnt fordelt på intervallet $[-5, 5]$. Om vi øker til 101 punkter får vi

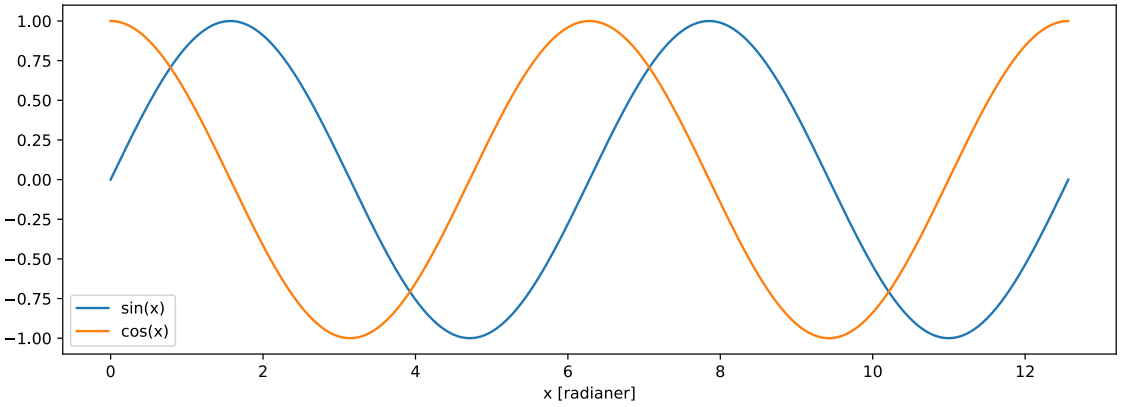
```
x = linspace(-5, 5, 101)
print(x)
```

```
[-5.  -4.9 -4.8 ... 4.7 4.8  4.9  5. ]
```

Ettersom at `linspace` også gir arrays kan den også brukes til å lage kjappe plots. La oss f.eks lage et plot med noen trigonometriske funksjoner

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

x = np.linspace(0, 4*pi, 1001)
y1 = sin(x)
y2 = cos(x)

plot(x, y1, label='sin(x)')
plot(x, y2, label='cos(x)')
legend()
xlabel('x [radianer]')
show()
```

Her plotter vi sinus og cosinus. Ettersom at `numpy` sine trigonometriske funksjoner er definert i form av radianer lar vi x gå fra 0 til 4π. Det koster lite for Python å regne ut mange verdier, så for å få en jevn og fin kurve velger vi like greit 1001 punkter. Resultatet er vist i Figur 12.



Figur 12: Plott av en sinuskurve og en cosinuskurve i samme figur.

9.8 Eksempel: Plott med parameterfremstilling

La oss se på et annet eksempel fra matematikken, å plotte kurver i planet ved hjelp av parameterfremstilling.

9.8.1 Rett linje

Først vil vi plotte den rette linja som knytter sammen de to punktene $A = (x_0, y_0)$ og $B = (x_1, y_1)$. Linja som knytter disse to punktene sammen er gitt ved parameterfremstillingen

$$\begin{cases} x = x_0 + t(x_1 - x_0), \\ y = y_0 + t(y_1 - y_0). \end{cases}$$

Vi kan da lage et generelt program som gjør dette for hvilke som helst to punkter A og B:

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

# Spesifiserer to punkter
A = (2, 3)
B = (7, -2)

# "Pakker ut" koordinatene
x0, y0 = A
```

```
x1, y1 = B

# Lager kurva
t = np.linspace(-1, 2, 101)
x = x0 + t*(x1 - x0)
y = y0 + t*(y1 - y0)

# Plot punktene
plot((x0, x1), (y0, y1), 'o')
plot(x, y)
show()
```

Merk her at vi spesifiserer A og B som verdipar med myke parenteser: `A = (2, 3)`. Dette er en variabeltype som kalles *tupler* i Python, det blir helt ekvivalent med å bruke lister (`A = [2, 3]`), vi bare liker at tuplene ligner litt mer på slik vi skriver punkter for hånd. Merk at vi pakker ut de to punktene til x- og y-koordinatene for å regne med disse verdiene. Om man syns den delen av koden var lite oversiktlig kunne vi istedet rett og slett definert de fire variablene x_0, y_0, x_1, og y_1 direkte. Resultatet av koden er vist i Figur 13.

9.8.2 Sirkel

Nå vil vi plotte en sirkel ved hjelp av parameterfremstilling. Man kan gjøre det ved å plotte kurva

$$\begin{cases} x = r \cdot \cos(\theta), \\ y = r \cdot \sin(\theta), \end{cases}$$

der r er sirkelens radius, og vinkelen θ går fra 0 til 2π.

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

r = 2
theta = linspace(0, 2*pi, 1001)

x = r*cos(theta)
y = r*sin(theta)

plot(x, y)
axis('equal')
```

```
show()
```

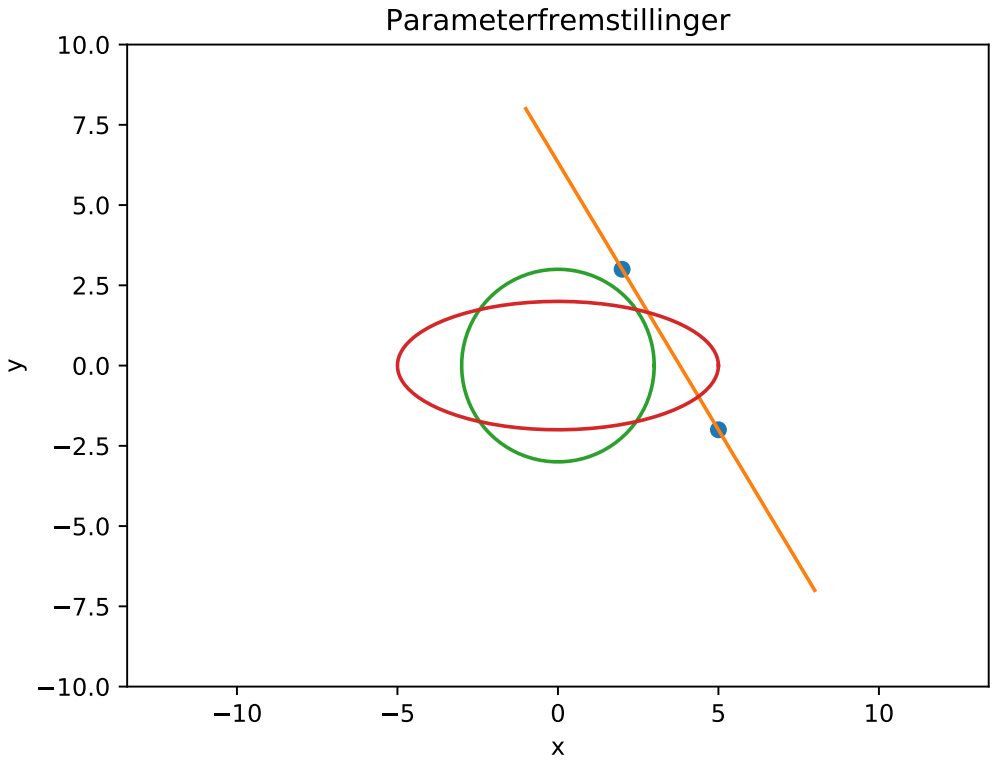
Her har vi lagt til linja `axis('equal')`, denne linja er strengt tatt ikke nødvendig, men om vi dropper den vil sirkelen vår fort se ut som en ellipse fordi x og y-aksen har ulike dimensjoner. Ved å si "axis equal" forteller vi Python at vi ønsker at de to aksene skal være dimensjonert likt, og sirkelen ser ut som en sirkel. Resultatet av koden er vist i Figur 13.

9.8.3 Ellipse

For å plotte en ellipse endrer vi bare på forrige eksempel så x og y verdiene er skallert med hver sin "radius", kalt "halvaksene" til ellipsen. Koden blir da:

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

a = 5
b = 2
theta = linspace(0, 2*pi, 1001)

x = a*cos(theta)
y = b*sin(theta)

plot(x, y)
axis('equal')
```

Denne er også vist i Figur 13



Figur 13: De tre parameterfremstillingene vist i samme figur.

9.9 Andre former for plotting

Matplotlib-pakken vi bruker til plotting har langt fler muligheter for å lage plott en kurveplott. Man kan for eksempel plotte strømlinjer, konturer, barplott, pai-diagrammer og mange andre muligheter. Derimot vil det for de fleste som er ferske i programmering være langt kjappere å produsere slik grafikk i annen programvare man er mer vant med, for eksempel GeoGebra eller Excel.

Det er i praksis ingen grense for hva man kan lage av plott og grafikk med Python. Det er derfor lite hensiktsmessig for oss å prøve å dekke mange ulike muligheter. Istedet vil vi nevne at de som står bak plottepakken har en egen nettside med eksempler der man kan se ulike figuren og koden som har lagd dem. Denne kan være nyttig om man lager et konkret undervisningsopplegg, og vil kunne lage et noe spesielt plott. Disse eksemplene finner du på

- https://matplotlib.org/gallery/

10 Sammensatt eksempel 3: Tilfeldighet

Vi dekker et siste tema før vi sier oss ferdig for denne gang, og det handler om hvordan vi kan innarbeide tilfeldighet i programmene våres. Tilfeldighet er et viktig element i spill, bare tenk på hvor mange brettspill som bruker terninger, eller stokking av kort eller lignende for å innarbeide tilfeldige elementer. Men tilfeldighet er også et viktig verktøy i matematikk, sannsynlighetsregning og statistikk.

For å få tilgang på tilfeldighet i koden vår kan vi bruke et tileggsbiblotek til Python som heter `random`. Navnet `random` er engelsk og betyr *tilfeldig*. Det finnes mange forskjellige funksjoner i dette biblioteket som man kan bruke til mye forskjellig. I dette kompendiet skal vi bare nevne to av dem.

10.1 Å lage tilfeldige tall

Den første funksjonen vi skal se på heter `randint`. Dette er et forholdsvis kryptisk navn, men det kommer fra "*rand*om *int*eger", som betyr "tilfeldig heltall" på engelsk. Denne funksjonen gir oss altså et tilfeldig heltall

Vi må altså først importere funksjonen ved å skrive

```
1 from random import randint
```

Om vi nå skriver `randint(a, b)`, der `a` og `b` er to tall, så vil funksjonen gi oss et tilfeldig tall i intervallet $[a, b]$. Si for eksempel at vi ønsker å rulle en terning, da kan vi skrive

```
1 from random import randint
2
3 print(randint(1, 6))
```

Hver gang dette programmet kjøres vil vi få et helt tilfeldig resultat mellom 1 og 6, altså tilsvarende det å rulle en terning. Vi kan også velge helt andre grenser.

Funksjonen `randint` er egentlig veldig enkel å bruke, men det kan være litt vanskelig å finne opp bruksområder. Vi går nå derfor igjennom to enkle spill man kan lage med hjelp fra denne funksjonen.

10.2 Eksempel: Gangetabellquiz

Vi har allerede sett på hvordan vi kan lage veldig enkle quizzer med if-tester. Problemet er kanskje at det er ikke så veldig morsomt å svare på sine egne quizzer. Nå skal vi istedet lage en mer matematisk quiz, men der spørsmålene genereres tilfeldig. Vi skal rett og slett lage en *gangetabellquiz*.

Som alltid går vi frem steg for steg. Vi fokuserer derfor først på å lage ett enkelt spørsmål. Først bruker vi `randint` for å trekke to tilfeldige tall mellom 1 og 10:

```
from random import randint

a = randint(1, 10)
b = randint(1, 10)
```

Spørsmålet til brukeren blir nå hva $a \times b$ blir. Vi bruker `input`-funksjonen til å stille dette spørsmålet, men siden vi skal sammenligne svaret med et tall må vi huske å gjøre om svaret fra brukeren til et tall.

```
(...fortsetter fra koden over)
svar = int(input(f"Hva er {a}x{b}?"))
```

Vi kan nå kjøre programmet og se om vi er på rett spor

```
Hva er 5x9? 45
```

Hver gang programmet kjøres så vil de to tallene endre seg.

Nå gjenstår det å sjekke om svaret brukeren har oppgitt stemmer. Dette gjør vi med en if-test:

```
if svar == a*b:
    print("Riktig!")
else:
    print(f"Feil. ({a}x{b} = {a*b}))
```

Her har vi lagt til en else-blokk som gir fasitsvaret dersom brukeren svarer feil.

Nå bør vi igjen kjøre programmet og sjekke at det fungerer som det skal. Da bør vi sjekke både et svar vi vet stemmer, og et vi vet er feil.

```
Hva er 8 x 1? 8
```

```
Riktig!

Hva er 8 x 7? 52
Feil. (8 x 7 = 56)
```

Når vi har kommet så langt at vi tester og ser at ett spørsmål fungerer fint, så er neste steg å legge til flere runder. Si for eksempel at vi skal svare på 5 oppgaver og holde styr på antall riktige.

Dette kan vi få til med en for-løkke som følger:

```
from random import randint

riktige = 0

for spørsmål in range(5):
        a = randint(1, 10)
        b = randint(1, 10)

        svar = int(input(f"Hva er {a} x {b}? "))

        if svar == a*b:
            print("Riktig!")
            riktige += 1
        else:
            print(f"Feil. ({a}x{b} = {a*b})")

print(f"Du fikk {riktige} av 5 rette.")
```

Her vil brukeren få 5 tilfeldige gangestykker, og til slutt får de skrevet ut en beskjed om hvor mange rette de hadde.

```
Hva er 9 x 7? 63
Riktig!
Hva er 4 x 9? 36
Riktig!
Hva er 2 x 10? 20
Riktig!
Hva er 3 x 8? 21
Feil. (3x8 = 24)
Hva er 2 x 6? 12
```

```
Riktig!
Du fikk 4 av 5 rette.
```

Kanskje vi også skal legge in en liten ekstrabeskjed helt til slutt om brukeren har fått alt helt rett?

```
if riktige == 5:
    print("Wow! Full pott! Du er jo helt rå!")
```

Utvidelser

Målet med denne øvelsen er å lage et lite spill og trene på ulike programmeringskonsepter. Men det kan jo også være et fint verktøy til å øve seg på gangetabellen.

På ungdomsskolen er det nok mange som allerede er godt drevne på gangetabellen, men har kan man jo justere programmet til det finner en passe vanskelighetsgrad. Kanskje man skal la begge tallene trekkes fra 1 til 15? Eller la et tall være mellom 1 og 10, men det andre mellom 1 og 20? Eller så kan man jo legge inn mer kompliserte regnestykker med andre operasjoner som for eksempel potenser. På denne måten kan elever justere til de får en passende utfordring.

Om man virkelig vil gjøre dette til et større prosjekt kan man tilogmed lage et program som setter opp små ligninger man må løse. Men det går vi ikke inn på nå.

En siste mulighet som kanskje kan være med å gjøre spillet litt mer spennende og konkuranseorientert er å legge til *tidtaking*. Dette har vi ikke dekket i kurset, men idéen er ganske grei. Fra bibloteket *time* kan vi importere en funksjonen ved samme navn. Denne funksjonen gir den nåværende tiden i sekunder (av årsaker vi ikke går inn på er det teknisk sett antall sekunder fra 1. januar 1970). Om vi sjekker tiden før og etter brukeren er ferdig med å svaret på alle spørsmålene vil differansen være tiden de brukte

```
from time import time

start = time()

# Resten av koden

slutt = time()
```

```
8 print(f"Du brukte {slutt-start:.1f} sekunder")
```

Etter vi har lagt til dette kan vi nå ha konkurranser. To personer kjører koden likt. Det er om å gjøre å få flest rette, men dersom begge klarer like mange er det den som var raskest som vinner.

```
Du fikk 10 av 10 rette.
Du brukte 13.4 sekunder.
```

10.3 Eksempel: Gjettespillet Over/Under

Et annet lite spill vi kan programmere er *Over/Under*, en enkel gjettelek man kan leke med to personer, men som også egner seg for programmering.

Frakoblet

Over/under lekes ved at man settes sammen i par. Person A tenker på et tilfeldig tall mellom 1 og 1000 og skriver det ned på et ark, så den andre personen ikke kan se det.

Deretter skal person B prøve å gjette seg frem til tallet. Etter hvert forslag B gir, så skal person A si om gjettet var over, under, eller helt riktig. På denne måten får B mer og mer informasjon for hvert gjett, og kan peile seg inn til riktig svar.

Etter at person B gjetter helt riktig tall, bytter man roller. Denne gangen skal B finne på et tall, og A gjette seg frem til det. Målet er å komme frem til riktig tall på færrest mulig gjett.

Om man leker dette i klasserommet kan man for eksempel la elevene leke tre runder hver. For hver runde bør de skrive ned hvor mange gjett de brukte. Den eleven som klarte å få til alle tre rundene på færrest gjett totalt er vinneren.

På datamaskin

La oss nå prøve å kode opp dette spillet. Først må vi la datamaskinen trekke et tall mellom 1 og 1000 tilfeldig. Dette kan vi gjøre som følger:

```
from random import randint

fasitsvar = randint(1, 1000)
print("Jeg har tenkt på et tall mellom 1 og 1000.")
print("Prøv å gjett det!")
print()
```

Merk datamaskinen her vil trekke et tilfeldig tall, men ikke skrive det ut, så det er hemmelig, akkurat som når man skriver ned tallet man velger på lappen som man ikke viser frem. Etter dette skriver vi ut instrukser til brukeren. Når vi skriver `print()` uten noen beskjed i parentesene så printer vi en blank linje.

Nå skal vi la brukeren gjette på et tall. Dette gjør vi med `input`, og som vanlig må vi huske å gjøre om svaret til et tall:

```
gjett = int(input("Gjett: "))
```

Nå skal vi begynne å sjekke om brukeren har gjettet riktig eller ikke, og da er kanskje første instinkt å bruke en if-test. Men nå er ikke målet at brukeren skal gjette bare én gang, men isteden å fortsette å gjette helt til de treffer rett svar. Når man skal gjenta en prosess helt frem til man når et gitt punkt er det en while-løkke som er mest naturlig. Derfor skriver vi

```
while gjett != fasitsvar:
    ...
```

Her skriver vi `!=`, fordi det betyr *ikke lik* (tenk `!=` ligner på $\neq$). Nå vil alt vi skriver inne i løkka bli gjentatt helt til brukeren svarer riktig. Når brukeren svarer riktig er løkka ferdig.

Vi kan nå begynne å fylle inn i løkka. Om brukeren har svart feil må vi sjekke om de er over eller under. Vi lager derfor en if-test:

```
while gjett != fasitsvar:
    if gjett > fasitsvar:
        print(f'Ditt gjett på {gjett} er for høyt.')
    if gjett < fasitsvar:
        print(f'Ditt gjett på {gjett} er for lavt.')
```

I tilegg må vi huske å la brukeren gjette på nytt, ellers ender vi opp med en uendelig løkke!

```
while gjett != fasitsvar:
    if gjett > fasitsvar:
        print(f'Ditt gjett på {gjett} er for høyt.')
    if gjett < fasitsvar:
        print(f'Ditt gjett på {gjett} er for lavt.')
    gjett = int(input('Prøv på nytt: '))
```

Det eneste som gjenstår er nå å legge til en endelig beskjed når brukeren til slutt svarer riktig. Denne legger vi rett og slett etter hele while-løkka. Hele programmet blir nå

```
from random import randint

fasitsvar = randint(1, 1000)
print("Jeg har tenkt på et tall mellom 1 og 1000.")
print("Prøv å gjett det!")
print()

gjett = int(input("Gjett: "))

while gjett != fasitsvar:
    if gjett > fasitsvar:
        print(f'Ditt gjett på {gjett} er for høyt.')
    if gjett < fasitsvar:
        print(f'Ditt gjett på {gjett} er for lavt.')
    gjett = int(input('Prøv på nytt: '))

print("Helt riktig!")
```

En kjøring av programmet gir nå for eksempel

```
Jeg har tenkt på et tall mellom 1 og 1000.
Prøv å gjett det!

Gjett: 400
Ditt gjett på 400 er for lavt.
Gjett: 700
Ditt gjett på 700 er for høyt.
Gjett: 600
Ditt gjett på 600 er for høyt.
```

```
Gjett: 500
Ditt gjett på 500 er for høyt.
Gjett: 450
Ditt gjett på 450 er for lavt.
Gjett: 475
Ditt gjett på 475 er for høyt.
Gjett: 465
Ditt gjett på 465 er for høyt.
Gjett: 455
Ditt gjett på 455 er for lavt.
Gjett: 462
Ditt gjett på 462 er for høyt.
Gjett: 459
Ditt gjett på 459 er for høyt.
Gjett: 457
Ditt gjett på 457 er for høyt.
Gjett: 455
Ditt gjett på 455 er for lavt.
Gjett: 456
Helt riktig!
```

Det kan være fint å legge til en ekstra detalj, nemlig at programmet til slutt kan skrive ut hvor mange gjettninger man bruker. Dette kan vi gjøre ved å lage en variabel før while-løkka, og så øke den med én for hver gang løkka kjører.

```
1  gjett = int(input("Gjett: "))
2  antall_gjett = 1
3
4  while gjett != fasitsvar:
5      ...
6      gjett = int(input("Prøv på nytt: "))
7      antall_gjett += 1
8
9  print("Helt riktig!")
10 print(f"Du bruke {antall_gjett} gjett totalt.")
```

Strategi

Bare det å kode opp over/under er en god øvelse for å lære programmering og algoritmisk tankegang, det kan også være morsomt. Derimot kan vi gå et steg lenger, og nå tenke på hvordan vi bør *spille* dette spillet for å gjøre det best mulig. Altså, hvilken strategi skal vi velge? Dette er også en god øvelse i algoritmisk tankegang!

Her kan vi for eksempel velge å bruke *midtpunktsmetoden*, der vi alltid velger midtpunktet av det intervallet vi har igjen. Da starter vi altså med å gjette 500. Om vi er for lave går vi opp til 750, om vi er for høye går vi ned til 250.

Midtpunktsmetoden er en god strategi, fordi uansett om vi er for høye eller lave kan vi eliminere halve det intervallet som er igjen. Et annet godt navn for denne strategien er *halveringsmetoden*.

Her kan det for eksempel være en god øvelse å regne seg frem til hvor mange gjett man trenger for å *garantert* komme frem til riktig svar med midtpunktsmetoden, som blir

$$\log_2(1000) = 9.9658...$$

Som betyr at med halveringsmetoden er vi *garantert* å komme frem til riktig svar på 10 gjett.

Her kan det nevnes at midtpunktsmetoden ikke bare er en god strategi for over-/under leken, men en god matematisk metode for å løse matematiske ligninger. Men dette er mer VGS pensum, så vi lar det ligge.

10.4 Å gjøre tilfeldige valg

En annen nyttig funksjon fra `random`-bibloteket heter `choice`, som er engelsk for "valg". Denne funksjonen gjør et tilfeldig valg fra en rekke muligheter for oss.

Si for eksempel at vi har 5 personer som er med i trekningen for et par kinobiletter. Her må vi *velge* én person tilfeldig. Det kan vi gjøre med `choice`.

For å bruke `choice` må vi bruke en *listevariabel*, disse har vi ikke dekket tidligere i dette kurset, men de er forholdsvis greie å jobbe med. Vi viser først eksempelet, så forklarer vi:

```
1 from random import choice
```

```
personer = ["Anders", "Beate", "Christine",
            "Daniel", "Erika"]
vinner = choice(personer)
print(vinner)
```

Første linje importerer `choice`-funksjonen. Den andre kodelinjen definerer en *liste* med personer. Her bruker vi firkantparenteser (`[]`) for å si at vi skal lage en liste, inne i lista deler vi hver ting i lista med komma. Her har vi en liste med 5 personer, så da bruker vi komma mellom hvert navn.

På den tredje linja skriver vi `choice(personer)`, det betyr at vi skal plukke ut én person tilfeldig fra list. Vi lagrer den personen som ble trukket i en variabel vi kaller `vinner`. Til slutt skriver vi ut vinneren.

Alle personene i lista har samme sannsynlighet for å bli valgt, og hver gang vi kjører programmet vil valget gjøres tilfeldig, og uavhengig av de andre kjøringene.

Dette var et veldig enkelt eksempel. Men la oss nå se på et litt mer komplisert tilfelle, nemlig stein-saks-papir.

10.5 Eksempel: Stein-Saks-Papir

Et klassisk spill å programmere opp er stein, saks, papir. Her tenker man seg kanskje at dette spiller ikke har noen *tilfeldighet* i seg. Men poenget er at vi skal lage spillet slik at man spiller mot datamaskinen, og da vil vi at datamaskinen skal velge stein, saks, og papir, tilfeldig.

Vi kan få datamaskinen til å velge ett av de tre alternativene med `choice`-funksjonen:

```
from random import choice
datamaskin = choice(["stein", "saks", "papir"])
```

Så må vi spørre brukeren om hva de velger. Da bruker vi `input`. Her er det litt viktig at de svarer et av de gyldige svarene nøyaktig, så vi skriver dem ut i spørsmålet:

```
bruker = input("Velg stein, saks, eller papir: ")
```

Etter spilleren har gjort valget sitt kan vi skrive ut de to valgene:

```
print(f"Du valgte: {bruker}")
print(f"Datamaskinen valgte: {datamaskin}")
```

Det som gjenstår da er å bruke if-tester til å finne ut hvem som vant. Her er det jo ganske mange muligheter, så vi må skrive en del kode, derimot må vi bare holde tunga rett i munnen, så kommer vi oss fort gjennom alle mulighetene.

```
if bruker == "stein":
    if datamaskin == "saks":
        print("Du vinner!")
    elif datamaskin == "papir":
        print("Du taper!")
    else:
        print("Uavgjort!")

elif bruker == "saks":
    if datamaskin == "papir":
        print("Du vinner!")
    elif datamaskin == "stein":
        print("Du taper!")
    else:
        print("Uavgjort!")

elif bruker == "papir":
    if datamaskin == "stein":
        print("Du vinner!")
    elif datamaskin == "saks":
        print("Du taper!")
    else:
        print("Uavgjort!")

else:
    print("Jeg skjønner ikke valget ditt, så jeg kan ikke
       kåre en vinner.")
```

Den siste `else`-blokken slår inn om brukeren ikke skrev inn et av valgene: stein, saks, eller papir, helt nøyaktig.

```
Velg stein, saks, eller papir: saks
Du valgte: saks
```

```
Datamaskinen valgte: saks
Uavgjort!

Velg stein, saks, eller papir: saks
Du valgte: saks
Datamaskinen valgte: papir
Du vinner!
```

Stein-saks-papir er en fin oppgave, fordi alle vet allerede godt hvordan man spiller spillet. Derimot kan det være en morsom utfordring å faktisk programmere det opp.

En annen grunn til at dette er en populær oppgave er at etterhvert som man lærer mer programmering kan man komme unna med langt færre if-tester og mindre kode, ved å gjøre smarte sammenligninger. Så det er en oppgave som er mulig å få til for nybegynnere, men etterhvert som man blir mer erfaren kan man finne mer elegante og effektive løsninger.

For de flinkeste elevene kan man også foreslå enkle utvidelser. For eksempel kan man modifisere spillet så man fortsetter å spille til enten du eller datamaskinen har fått 3 poeng. For å gjøre dette vil en while-løkke være et naturlig valg.

11 Tillegg A—Printing

For å flette variabler inn i tekststrenger skriver vi `f` foran strengen (tenk *f* for *f*letting). Da kan vi skrive variabelen rett inn i teksten med å skrive den på innsiden av krøllparenteser (`{}`).

```
1 navn = "Mari"
2 print(f"Hei på deg {navn}!")
```

```
Hei på deg Mari!
```

Dette er alt man egentlig trenger å vite for å lage programmer og løse de fleste interessante problemer. Derimot kan det være greit å kjenne til at man kan finjustere detaljer som antall desimaler som skal skrives ut, man kan skrive tall på standardform osv. Dette trenger man ikke nødvendigvis å undervise til elever, men heller informere om dersom noen etterspør denne informasjonen.

For å finjustere hvordan en variabel skrives ut skriver vi kolon ( ) bak variabelen. Om vi skriver et tall rett bak kolonet bestemmer det bredden på utskriften, så vi kan lage fine kolonner:

```
for tall in range(5):
    print(f"{tall:5}{tall**2:5}{tall**3:5}")
```

```
    0    0    0
    1    1    1
    2    4    8
    3    9   27
    4   16   64
```

I tilegg kan vi skrive punktum og et tall for å spesifisere antall desimaler som skal skrives ut. Da må vi også spesifisere en type. Så vi kan for eksempel skrive π til 5 desimaler som følger:

```
from math import pi
print(f"{pi:.5f}")
```

```
3.14159
```

Under er en kort tabell som viser de vanligste typene vi velger for utskrift:

`f`	Desimaltall (f står for flyttall)
`e`	Standardnotasjon
`g`	Vanlig desimal for små tall, standardnotasjon for store
`%`	Tallet ganges med 100 %

For eksempel:

```
rente = 0.0345
print(f"Kontoen har en årlig rente på: {rente:.3\%}")
```

```
Kontoen har en årlig rente på: 3.450\%
```

12 Tillegg B—Begrepsliste

Å skulle lære koding og programmering handler om mye mer enn bare det å skrive kode. Enda viktigere er tenkemåten og fremgangsmåtene man bruker for å bryte

ned og løse problemer. Det betyr også at det er viktig med et godt begrepsapparat, slik at man lærer seg å *snakke kode*, og kunne diskutere seg frem til løsninger.

Her har vi en kort liste med noen viktige begreper og forklaringene av dem. Dette er på ingen måte en komplett liste, men dekker en del viktige begreper det er fint å inkludere i undervisningen.

Algoritmer Dette er oppskrifter på hvordan man skal gå frem for å løse et gitt problem. Algoritmer skal være en helt presis og konkret beskrivelse av stegene man skal gjøre, for å komme frem til et konkret resultat. Et klassisk eksempel er stegene i en kakeoppskrift.

Algoritmer er viktig i alle former for problemløsning, også for eksempel i mattetimen. De er derimot også spesielt viktige innen programmering, fordi datamaskiner er egentlig ganske dumme og kan derfor kun forstå veldig enkle instrukser. For å programmere må vi altså forstå hvordan vi kan ta et komplekst, sammensatt problem, og bryte det ned i små, enkle steg som datamaskinen kan forstå. Dette kalles gjerne for *algoritmisk tankegang*.

Programmer Et program er et sett med instrukser til datamaskinen om å gjøre en oppgave eller å løse et problem. Mer konkret er det en fil på maskinen som inneholder kode som datamaskinen forstår. Vi sier at vi *kjører* et program, og da utøver maskinen koden i programmet. Små programmer, slik de vi skriver, kalles gjerne for *scripts*. Et alternativ til å si at vi programmerer, eller koder, er derfor å si at man *scripter*.

Programmeringsspråk Dette er språk som datamaskinen forstår. I likhet med vanlig språk finnes det mange forskjellige programmeringsspråk, og til og med dialekter av språk. Disse språkene kan enten være *blokkbaserte*, eller *tekstbaserte*. Python er et eksempel på et tekstbasert programmeringsspråk. Når vi skriver Python skriver vi ut kode som en vanlig tekst.

Kodelinjer Én enkelt linje med kode i et helt program. Hver kodelinje er en instruks til datamaskinen om å gjøre en viss handling. Når vi skal løse en gitt problemstilling med programmering må vi først bryte problemet ned i enkelte steg, altså lage en algoritme, og så må vi skrive disse stegene som kodelinjer. På engelsk

kalles en kodelinje for en *statement*. Et alternativ på norsk kan derfor kanskje være *setninger* eller *kodesetninger*.

Variabler Dette er slik datamaskinen kan huske på informasjon, og variabler er viktige verktøy når man programmerer. Variabler i programmeringsverden har mye til felles med variabler i matematikken. Å forstå begrepet *variabel* og bruke det korrekt er en viktig del av det å lære matematikk *og* programmering.

Løkker Vi bruker løkker for å repetere steg i en algoritme. Vi bruker løkker for å forenkle koden vår, men også for å generalisere og forbedre. Datamaskiner kan kun utføre veldig små og enkle oppgaver, men de kan til gjengjeld gjøre dem veldig, veldig fort. Løkker er derfor en god måte å lage algoritmer der datamaskinen kan gjenta enkle steg veldig mange ganger. Et annet ord for løkker er *sløyfer*.

Tester En test er en måte for datamaskinen og sjekke en *betingelse*, og gjøre forskjellige ting basert på konteksten. På engelsk kalles dette gjerne for *if*-tester, som kan oversettes med *hvis-så* setninger. Man kan også kalle dem for *betingelsessetninger*. Tester er viktig, fordi det er slik vi kan innarbeid logikk inn i programmene våres. I motsetning til oss mennesker kan ikke en datamaskin jobbe utifra kontekst, og vil derfor måtte ha helt konkrete betingelser som skal bestemme oppførselen dens.

Funksjoner Dette er regler vi *definerer*. Funksjoner er igjen et begrep som er viktig i både programmering og i matematikk. I programmeringen brukes funksjoner til å forenkle programmene våres og gjenbruke kode. De gjør det også lettere å jobbe med *abstraksjon* av idéer. Ved å lære å programmere funksjoner kan matematiske funksjoner bli mer håndfast for noen elever. Felles for alle funksjoner er at de har en konkret definisjon på hvordan de fungerer og oppfører seg når vi bruker dem.