Introduksjonskurs i programmering

En kort innføring i programmering og Python

kodeskolen simula

kodeskolen@simula.no

Dette kompendiet er kursmateriale for et kurs i programmering for deg 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, og hvordan man kan bruke det til å automatisere enkle, repetitive oppgaver. 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. Vi prøver derimot 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 oppgaver og prosjekter man kan få til ved hjelp av programmeringen.

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 kolleger, 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 diskuterer 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, og er interessert i svaret på dette problemet, men fremgangsmåten kan også være interessant i seg selv.

Programmering er i likhet med matematikk basert på logikk. Matematikk har vært viktig for å utvikle viktige verktøy innen programmering, og programmering har vært viktig for å løse matematiske problemer som ikke kan løses for hånd. Ved å kombinere matematikk og programmering er det for eksempel mulig å forstå dynamikken i fysiske, biologiske, geologiske og kjemiske systemer, forutsi hvordan for eksempel været vil være de neste dagene, eller hvordan verdensøkonomien vil utvikle seg de neste månedene. Imidlertid er ikke all programmering nært knyttet til matematikk – det brukes også til bildeanalyse, til å utvikle programvare, nettsider og mye annet.

All programmering bygger på felles grunnkonsepter – variabler, betingelser, løkker, funksjoner, datastrukturer – og i dette kompendiet får du en introduksjon til hver av disse. Videre skal vi gå igjennom enkel filhåndtering. Vi håper dette vil fremstå som nyttige verktøy for databehandling og problemløsning.

Du som leser dette skal kanskje ikke løse vanskelige ligninger, forutsi hvordan aksekursen kommer til å svinge eller utvikle avansert programvare. Men kanskje kan du jobbe med databehandling – fra filer eller fra internett. Du kan også kombinere

dette med statistikk, slik at man får viktige nøkkeltall. Og hvis du synes det er gøy, kan du lage kule gadgets ved hjelp av Mikro:Bit eller Raspberry Pi.

Uansett håper vi at du vil synes at programmering er gøy, at du kan se hvordan det kan være et nyttig verktøy og at du kan bruke det du har lært her som grunnvoll til å lære mer, uansett hvilken retning du ønsker å gå i.

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.

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?

1.2.2 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?
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.

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).

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.

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 fordi det anses som nybegynnervennlig, og det er plattformuavhengig. Man kan derfor kjapt komme igang med enkle programmer og eksempler.

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 Python-program, hvor vi forklarer steg for steg hvordan du skriver kode, lagrer programmet, kjører det og ser resultatet.

Dette kapittelet 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 Python-programmering mer detaljert.

2.1 Ditt første Python-program

For mer informasjon om hva slags programvare man kan bruke til å programmere i Python, se tillegg A, bak i kompendiet.

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

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 vil 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 har 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 et 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 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 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 i kapittel 8.

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.

2.1.1 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):

```
1 print('''Liten?
2 Jeg?
3 Langtifra.
4 Jeg er akkurat stor nok.
5 Fyller meg selv helt
6 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?
```

2.1.2 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\_//
 |  ___  \_/\---'
 |_||  |_||
''')
```

2.1.3 Variabler og formatering

For å ta vare på informasjon i programmering bruker vi variabler – dette skal vi lære mer om i kapittel 3. For nå kan dere tenke på disse som matematiske variabler.

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 (`{}`).

```
navn = "Mari"
```

```
print(f"Hei på deg {navn}!")
```

```
Hei på deg Mari!
```

Vi kan også skrive ut tall på denne måten:

```
gjennomsnittsfart = 43.33335231
print(f"Gjennomsnittsfarten er {gjennomsnittsfart} km/t.")
```

– som vil gi oss følgende output:

```
Gjennomsnittsfarten er 43.33335231 km/t.
```

Dette var med mange desimaler – kanskje unødvendig mange i dette tilfellet. For å finjustere hvordan en variabel som representerer et tall skrives ut kan vi legge til et kolon ( ) bak variabelen, sammen med et punktum, et tall for å spesifisere antall desimaler som skal skrives ut og en «f» (vi kommer tilbake til hva f betyr i 3). For å få med 2 desimaler kan vi for eksempel skrive det slik:

```
gjennomsnittsfart = 43.33335231
print(f"Gjennomsnittsfarten er {gjennomsnittsfart:.2f} km/
    t.")
```

– som vil gi oss

```
Gjennomsnittsfarten er 43.33 km/t.
```

Om vi skriver et tall rett bak kolonet bestemmer det bredden på utskriften, så vi kan også 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
```

Her har vi brukt noe som heter for-løkke for å skrive ut flere linjer – dette vil dere lære mer om i kapittel 7.

2.2 Avansert eksempel: Å regne ut volum

«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 bruker. Nå går vi igjennom et litt større eksempel for å vise hvordan et Python-program kan se ut. Mange av detaljene i dette programmet vil forklares bedre i det neste kapittelet.

```
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 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 volum, men Python kjenner ikke til denne fra før. Derimot kan vi, som vi gjør her, importere den fra matematikkbiblioteket `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 vare 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.

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.

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.

3.1 Opprette variabler

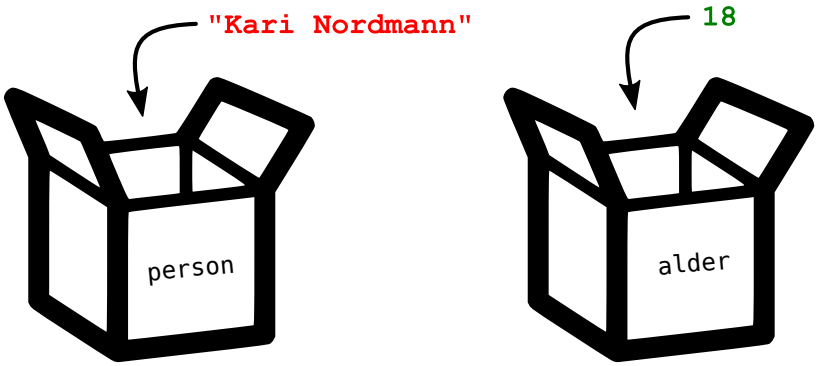
Vi oppretter variabler, ved å bruke likhetstegn (=), vi kan for eksempel skrive:

```
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 er 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 den frem igjen når vi ønsker den, takket være navnet.



Figur 2: 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.

3.1.1 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 direkte med en print-kommando, slik:

```
1 person = "Kari Nordmann"
2 print(person)
```

som vil gi

```
Kari Nordmann
```

eller vi kan flette den inn i en tekststreng:

```
1 print(f"Hei på deg, {person}")
```

```
Hei på deg, 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:

```
person = "Ola"
alder = 17

print(f"Gratulerer med dagen, {person}")
print(f"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 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.

3.1.2 Vanlige feilmeldinger

Når du starter – og når du fortsetter – å programmere, vil du garantert gjøre små feil som enten gjør at programmet ditt ikke kjører, eller at det kjører, men gir «feil» output i forhold til det du forventet. Feilmeldinger er ikke farlige. Det er bare datamaskinens måte å si ifra at «nå skjønner jeg ikke helt hva du mener» på. Videre gir den informasjon om hva den ikke skjønner – og dette er informasjon som vi kan bruke for å rette opp i programmet slik at det fungerer som det skal.

En vanlig feil som kan oppstå, er at man prøver å bruke en variabel før den er blitt opprettet eller definert. Dersom vi prøver å kjøre koden

```
print(f"Gratulerer med dagen, {person}")
```

Får vi feilmeldingen

```
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    print("Gratulerer med dagen, {person}!")
NameError: name 'person' is not defined
```

La oss gå gjennom hva denne beskjeden forteller oss.

- Den første linja sier at programmet går bakover og ser etter feil
- på linje to får vi beskjed om at feilen er i programmet "test.py", og at den er på linje 1.
- På tredje linje står det det som står i koden på stedet der feilen er.
- På siste linja står det hva datamaskinen mener at feilen er, nemlig at det er en navnefeil, og at variabelen `person` ikke er definert.

Denne informasjonen er nok til at vi forstår at før vi kan printe ut `person`, så må vi opprette variabelen først.

En annen vanlig feil, er å glemme å lukke en parentes, eller glemme å ha fnutter på begge sider av en streng.

```
1 person = "Jonas"
2 print(f"Gratulerer med dagen, {person}!
```

```
  File "test.py", line 3

                                          ^
SyntaxError: unexpected EOF while parsing
```

Denne beskjeden er litt annerledes. Den sier at vi har en `SyntaxError`, som betyr at det er noe i koden vår som ikke er godkjent python-syntaks. I dette tilfellet er det at alle parenteser må lukkes. Legg merke til at den sier at feilen er på linje 3, selv om den egentlig er på linje 2. Dette er ganske vanlig, og det kan ofte være lurt å titte både på linja over og linja under der datamaskinen påstår at feilen er.

3.1.3 Logiske feil

En tredje vanlig feil er å glemme å skrive «f» foran strengen. Isåfall vil koden fortsatt kjøre, men den vil ikke gi oss utskriften vi forventer. Koden

```
1 person = "Jonas"
2 print("Gratulerer med dagen, {person}!")
```

vil kjøre og skrive ut

```
Gratulerer med dagen, {person}!
```

Dette kalles for en *logisk feil*. Dette er en feil som Python ikke vil gi oss beskjed om, siden det faktisk *kan* være riktig – hvis vi ville ha en utskrift med krøllparanteser. Her må man selv være obs på sin egen logikk, og passe på at man spør datamaskinen om å utføre riktig operasjon.

3.2 Printing av variabler

Når man programmerer er det veldig nyttig å skrive ut informasjon. Åpenbart er det nyttig å skrive ut resultatet av et program til slutt – men det er også et uvurderlig verktøy for å følge med på hva som skjer underveis i programmet. Ikke minst er det et nyttig verktøy for å oppdage logiske feil.

Du har sett at vi kan skrive ut variabler enten for seg selv, eller kombinert med en tekststreng. Vi kan også skrive ut flere variabler på samme linje. Enten skiller vi de ved et komma (ved direkte utskrift), eller så refererer vi til de løpende i teksten:

```
1 person = "Kari Nordmann"
2 alder = 18
3 print(person, alder)
4 print(f"Hei! Jeg heter {person} og er {alder} år gammel.")
```

Når vi skriver ut tall, særlig for å vise frem et resultat, kan det være nyttig å formatere det på ulike måter. I introduksjonen så vi et eksempel på hvordan man skriver ut flyttall, og vi kan gjenta syntaksen for dette her. Hvis man vil skrive ut et desimaltall med to desimaler kan man skrive et kolon, et komma, et totall og en «f»:

```
1 gjennomsnittsfart = 43.33335231
2 print(f"Gjennomsnittsfarten er {gjennomsnittsfart:.2f} km/
    t.")
```

som vil gi

```
Gjennomsnittsfarten er 43.33 km/t.
```

Når vi skriver «f» angir vi en *type* utskrift; her angir vi at utskriften skal gis som et desimaltall. «f» står det for «floating number», dvs «flyttall» på norsk – informatikeres måte å si «desimaltall» på.

Noen ganger, særlig i matematisk og naturfaglig sammenheng, vil man helst bruke standardform for å skrive ut tallene. La oss se på jordens volum, som ifølge Wikipedia er

$$1.083207319801 \cdot 10^{12}$$

og massen av et proton, som er

$$1.67262192369 \cdot 10^{-27}.$$

For å skrive tall på standardform i Python bruker man `E` eller `e` (begge kan brukes, ingen forskjell). Dette vil også være slik utskriften av store/små tall vil se ut. For eksempel kan vi skrive ut Jordens volum og (omtrentlig) massen av et atom, der vi velger antall signifikante sifre som skal vises:

```
jordvolum = 1.083207319801e12
print(f"Jordens volum er {jordvolum:.13g} km^3, eller
   omtrent {jordvolum:.2g} km^3.")

protonmasse =   1.67262192369e-27
print(f"Et proton veier {protonmasse:.13g} kg, eller
   omtrent {protonmasse:.2g} kg.")
```

og dette vil skrives ut som følgende:

```
Jordens volum er 1083207319801 km^3, eller omtrent 1.1e+12
    km^3.
Et proton veier 1.67262192369e-27 kg, eller omtrent 1.7e
   -27 kg.
```

Her har vi angitt g som type; skriver man dette etter variabelen vil man få desimalform for små tall, og standardform for store/små tall.

Vi kan også skrive at vi vil skrive ut et tall i prosent:

```
rente = 0.0345
print(f"Kontoen har en årlig rente på: {rente:.3 %}")
```

og dette vil skrives ut som følgende:

```
Kontoen har en årlig rente på: 3.450 %
```

Her har vi angitt % som type; skriver man dette ganges tallet med 100 % før utskrift.

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 tillegg 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:

```
1 navn = input('Hva heter du? ')
2
3 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

```
1 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

```
1 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

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

for å ta vare og huske på svaret fra brukeren.

3.4 Typer

3.4.1 Ulike typer variabler

Vi har snakket om at en variabler har et *navn*, og en *verdi*. I tillegg til disse, har en variabel også en *type*.

Når vi har lagret variabler som er navn gjør vi dette med fnutter (') rundt navnet, det er fordi vi lagrer det som en variabel som har typen *tekst*. Variabler som alder har vi lagret uten disse fnuttene, fordi denne variabelen er et tall.

En tekstvariabel kalles i programmeringsverden for en *streng* (string på engelsk.)

Et heltall har typen *int*, og et desimaltall har typen *float*. Vi kan se typen til en variabel ved hjelp av kommandoen `type()`.

```
navn = 'Kari Nordmann'
alder = 18
pi = 3.14
print(type(navn))
print(type(alder))
print(type(pi))
```

```
<class 'str'>
<class 'int'>
<class 'float'>
```

String `str()` i Python, er typen som representerer tekst, eller strenger i programmeringssammenheng. Disse kan inneholde så mye tekst som man ønsker, hele bøker om man vil.

Integer `int()` i Python, er typen som representerer heltall. Som navnet skulle tilsi så er det kun hele tall som kan representeres, men disse kan være både positive og negative.

Float `float()` i Python, er typen som representerer desimaltall, eller flyttall som det heter i programmeringssammenheng. Dette er hvordan alle tall som har desimaler representeres. Tallene kan være positive og negative, men veldig små tall kan bli unøyaktige når de legges sammen.

3.4.2 Kasting mellom typer

Etterhvert som du lærer mer programmering, vil du til tider oppleve at en variabel du har laget eller hentet med `input()` har feil type. Da trenger vi å endre typen, noe som kalles *kasting*.

`input()`-funksjonen vil alltid lagre en variabel som en streng. I neste delkapittel skal vi se mer på regning i Python, men vi kan ta et eksempel med pluss her, for å vise hvorfor det er viktig å passe på riktig type.

Si at du ber om et tall med input, og printer ut det tallet plusset med seg selv.

```
tall = input('gi meg et tall: ')
print(tall+tall)
```

```
gi meg et tall: 3
33
```

Vi ga tallet 3, og fikk at 3+3=33, noe som er feil. Det er fordi programmet ikke har lagret `tall=3`, men `tall='3'`. Når du plusser sammen to strenger, vil Python bare legge dem etter hverandre.

For å få riktig svar må vi kaste `tall` til typen `'int'`, altså et heltall. Det gjør vi ganske enkelt ved å skrive kommandoen `int()` rundt det vi vil kaste til et heltall, som i dette eksempelet er inputen.

```
tall = int(input('gi meg et tall: '))
print(tall+tall)
```

```
gi meg et tall: 3
6
```

Det var bedre!

På tilsvarende måte kan vi kaste variabler slik at de blir strenger ved hjelp av kommandoen `str()`, eller kaste til desimaltall ved kommandoen `float()`. (NB: Legg merke til at python bruker punktum som desimaltegn, og ikke komma!)

Når vi kaster et desimaltall til et heltall, vil python ikke runde til nærmeste, men bare legge vekk desimalene til tallet.

```
print(int(3.825))
```

```
3
```

3.5 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 for eksempel fint bruke Python som en 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 tillegg 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:

```
print(volum)
```

```
5.5724533333333355
```

Dette er verdien Python har regnet ut for oss. Den er imidlertid 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 tillegg legge til litt tilleggsinfo:

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

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

Når vi skriver kodelinjen

```
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.

3.6 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 programmerings-

språk. 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

```
from math import sqrt
```

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

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

```
9.0
```

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

```
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.

3.6.1 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
```

3.6.2 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
```

3.7 Eksempel: 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 for å løse noen enkle mattestykker for oss. Vi må huske på å konvertere input fra tekst så vi kan gjøre matematiske operasjoner med det.

```
1 a = int(input('Skriv inn det første tallet: '))
2 b = int(input('Skriv inn det andre tallet: '))
3
4 print(f'A + B = {a + b}')
```

```
Skriv inn det første tallet: 6
Skriv inn det andre tallet: 7
A + B = 13
```

Det er verdt å merke at det her er brukt `int()` for konverteringen til *heltall*. Dette gjør at vi kun kan regne med heltall, da tall med desimaler ikke vil bli konvertert.

Om vi istedet ønsker et desimaltall, bruker vi `float()`.

En vanlig feil som kan oppstå, er at vi glemmer å kaste en eller flere variabler om fra streng til tall.

```
1 a = int(input('Skriv inn det første tallet: '))
2 b = input('Skriv inn det andre tallet: ')
3
4 print(f'A + B = {a + b}')
```

Da vil vi få følgende feilmelding

```
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    print(f'A + B = {a + b}')
TypeError: unsupported operand type(s) for +: 'int' and '
   str'
```

Denne feilmeldingen sier altså at det ikke er mulig å bruke + operatoren mellom to variabler som har type 'int' og 'str'. Når vi ser denne, er det ofte lurt å sjekke inputene, og se om vi har husket å kaste alle til tall.

Som et lite eksempel på der helt enkel utregning kan gi et program med «mening», kan vi regne ut hvor gammel noen er basert på når de er født.

Det kan vi gjøre ved for eksempel å skrive:

```
1 år = 2020
2 fødselsår = int(input('Hvilket år ble du født i? '))
3
4 print(f'Da blir du {år - fødselsår} år i år!')
```

```
Hvilket år ble du født i? 1997
Da blir du 23 år i år!
```

3.8 Avansert 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 konkretisere og sette opp hvordan man løser små problemer. Samtidig er de meget god øving i programmering og gode måter å forstå 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 at vi 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.

4 Betingelser

Det neste programmeringskonseptet vi skal se på er *betingelser*. Betingelser er viktige fordi det er slik vi innarbeider logikk i programmene våre. 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.

På engelsk kaller man gjerne betingelser for *if sentence*. Ordet «if» er «hvis» på norsk, så derfor kan vi kalle dem «hvis-setninger».

Når vi bruker betingelser anvender vi boolsk logikk. 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*.

4.1 Å skrive betingelser i Python

Den enkleste formen for en betingelse, 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 (også kalt indentering). 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

```
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:

```
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 Eksempel: Quiz

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

```
riktige = 0

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

```
4  if svar == "Lisboa":
5      print("Det stemmer!")
6      riktige += 1
7  else:
8      print("Det er feil. Riktig svar er Lisboa.")
9
10 svar = input("Hva slags dyr er Scooby Doo? ")
11 if svar == "hund":
12     print("Det stemmer!")
13     riktige += 1
14 else:
15     print("Det er feil. Scooby Doo er en hund.")
16
17 print(f"Det var det! Du fikk til {riktige} av 2 spørsmål."
       )
18
19 if riktige == 2:
20     print("Veldig bra jobba!")
```

Dette er et kort eksempel med bare to spørsmål, men her kan man enkelt utvide quizen og legge til flere spørsmål. For hvert spørsmål sjekker vi svaret og skriver eventuelt ut riktig svar om brukeren tar feil. I tillegg 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.

4.1.3 Vanlige feil

En vanlig feil som kan forekomme med if-else betingelser er indenteringsfeil. Disse er ikke alltid like lette å finne som andre typer feil, siden det ikke nødvendigvis er en egen feilmelding for disse. Hvis man glemmer å indentere i det hele tatt får man feilmeldingen:

```
1 if betingelse == True:
2 print('Dette er sant!')
```

```
  File "<stdin>", line 2
```

```
    print('Dette er sant!')
        ^
IndentationError: expected an indented block
```

Men koden kan ha feil uten å gi en feil melding, ved for eksempel å printe ut ting som bare skulle printes hvis det var en betingelse som var usann.

```
if betingelse == True:
    print('Dette er sant!')
else:
    c = a + b
    print(f'c har verdien {c}')
print('Dette er usant!')
```

```
betingelse = True
Dette er sant!
Dette er usant!
```

Man kan også få en annen feilmelding hvis man glemmer et kolon (:).

```
if betingelse == True
    print('Dette er sant!')
```

```
 File "<stdin>", line 1
    if betingelse == True
                          ^
SyntaxError: invalid syntax
```

4.2 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 betingelsene:

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

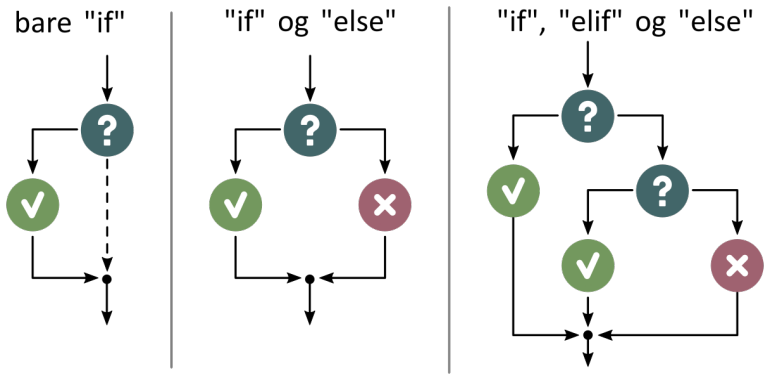
Merk at hvordan *større enn* og *mindre enn* tolkes avhengig av hva slags type variabler vi snakker om. For tall er det jo ganske greit, men hva med for eksempel tekst? Her vil Python bruke den alfabetiske sorteringen, slik at for to tekststrenger vil `a < b` være sant hvis `a` ville blitt sortert før `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 tillegg kan vi bruke `and` og `or` for å kombinere to eller flere betingelser. Forskjellen er at om vi bruker `and` vil den samlede betingelsen være sann, kun om begge betingelsene er sanne, mens om vi bruker *or* trenger bare en av betingelsene å være sanne for at den samlede betingelsen skal være sann.

Disse betingelsene og måtene å kombinere dem på kalles gjerne bolsk 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 bolsk logikk riktig.

En type feil som er lett å gjøre med logiske operatorer er å skive '=' når man for eksempel mener '=='. Dette vil gi en SyntaxError:

```
1 if a = b:
```



Figur 3: Tre forskjellige typer betingelser. 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 ønske.

```
    print('This is true!')
```

```
  File "<stdin>", line 1
    if a = b
         ^
SyntaxError: invalid syntax
```

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 betingelsen. 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, som i et flytdiagram.

4.4 Avansert eksempel: Flere typer muffins

I avsnitt 3.8 viste vi et eksempel på et program som skalerte en muffinsoppskrift utifra hvor mange muffins brukeren ønsket å lage. La oss nå inkludere betingelser 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-betingelse 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. Vi har imidlertid 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 3.8, 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.

5 Datastrukturer

Til nå har vi sett på enkeltvariabler, som har et navn, et innhold og en type. Etterhvert som vi lærer mer programmering, og begynner å lage større, mer kompliserte programmer, kan vi få mange variabler å holde styr på samtidig. Ofte kan det være hensiktsmessig, å samle disse i en felles større variabel, som tar vare på informasjonen vi vil lagre, uten å måtte opprette en variabel for hver lille ting.

Datastrukturer er en måte å samle mer data på ett sted når vi programmerer. Det finnes ulike varianter, og hvordan de fungerer kan variere noe fra ett programmeringsspråk til et annet. I dette kapittelet skal vi se på noen av de mest brukte, spesielt *lister* og *oppslagsverk*.

5.1 Lister

La oss introdusere en mye brukt datastruktur: lister. La oss si at du ikke bare ønsker at programmet ditt skal huske på et navn, men alle ansatte i en liten bedrift. Det er veldig slitsomt å måtte opprette en variabel for hver enkelt ansatt. Det vi kan gjøre, er å opprette en enkelt variabel, hvor vi lagrer alle ansatte sammen, det kan du gjøre sånn her

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

Vi ser at vi har brukt firkantparenteser, `[` og `]`, for å definere en liste (disse parentesene kalles også gjerne for *klammeparenteser*). 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(ansatte)
```

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:

```
1 print(len(ansatte))
```

```
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 ansatte

```
ansatte = ["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(ansatte[0])
print(ansatte[3])
```

```
Alma
Mohammad
```

Her så betyr `ansatte[0]` det første elementet i lista, som altså er 'Alma', mens `ansatte[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 `ansatte[4]` vi må endre. Da skriver vi

```
ansatte[4] = "Alexander"
```

Hvis vi nå skriver ut hele lista på nytt med `print(ansatte)` 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 ansatt, da kan den personen legges til som følger

```
ansatte.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.

En vanlig feil når det kommer til lister er såkalt ½off-by-one»-, eller «bommer med en»-feil. Dette er feilen vi mennesker lett kan gjøre ved å glemme oss når det kommer til at lister starter å telle på 0. I eksempelet under har vi en liste med elementene 1 til 9. Vi ønsker å hente ut tallet 7, så vi skriver `liste[7]`. Mens vi får et tall ut, så er det ikke tallet vi ønsker vi får tilbake. Denne «bommer med en»-feilen er lett å gjøre, og ikke alltid lett å oppdage. Her er det viktig å skrive ut det man gjør, slik at man oppdager eventuelle feil!

```
liste = [1,2,3,4,5,6,7,8,9]
print(liste)
print(liste[7])
```

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

En annen feil som kan forekomme med lister er en såkalt IndexError , eller "index-feil". Dette er en feil som kommer av at vi prøver å se på et element i en liste som ikke eksisterer. Hvis vi tar listen fra tidligere igjen, så ville dette oppstå ved å koble opp mot `liste[9]`. Vi vil komme mer inn på hvordan dette kan forekomme i kapittel 7 om Løkker(s.64).

```
liste = [1,2,3,4,5,6,7,8,9]
print(liste[9])
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

5.2 Andre listemetoder

Vi har sett på hvordan vi kan bruke `append()` til å legge til elementer i en liste, og `len()` til å finne ut hvor mange elementer den har. Dette er innebygde funksjoner, eller *metoder*, som lister har. Lister har også mange andre metoder som kan være nyttig å vite om. Her skal vi gå gjennom et par av dem. Du kan også lese mer om listemetoder her: https://docs.python.org/3/tutorial/datastructures.

reverse

`reverse()` reverserer rekkefølgen på elementene i lista, slik at det siste elementet blir først osv.

```
navn = ['Anne','Henrik','Vidar','Sofie']
navn.reverse()
print(navn)
```

```
['Sofie', 'Vidar', 'Henrik', 'Anne']
```

in

`in` operatoren lar oss sjekke om et element eksisterer i en liste. Dette er en test på om noe er sant/usant (True/False), og forteller oss ingenting om hvor i listen et element ligger, heller ikke hvor mange elementer som matcher det er i listen.

```
navn = ['Anne','Henrik','Vidar','Sofie']
'Anne' in navn
```

```
True
```

count

`count()`-metoden brukes på en liste for å se hvor mange elementer som er i en liste som matcher det man forer metoden. Som med `in` så gir det oss ingen informasjon om hvor i listen elementene ligger.

```
frukt = ['eple','banan','kiwi','appelsin']
frukt.count('eple')
frukt.append('eple')
frukt.count('eple')
```

```
1
2
```

sort

`sort()`-metoden brukes på en liste for å sortere den. Den sorterer tall etter størrelse, og tekst alfabetisk.

```
frukt = ['eple','banan','kiwi','appelsin']
tall = [10,3,1,12]
frukt.sort()
tall.sort()
```

```
5 print(frukt)
6 print(tall)
```

```
['appelsin', 'banan', 'eple', 'kiwi']
[1, 3, 10, 12]
```

Dersom vi ikke ber om annet, vil den sortere i stigende rekkefølge, men vi kan spesifisere at vi vil ha den i reveres

```
1 tall.sort(reverse=True)
```

```
[12, 10, 3, 1]
```

Flere typer indeksering

Til nå har vi brukt `min_liste[2]` for å hente ut element nr 2. I tillegg til denne indekseringen, kan vi også bruke

- `min_liste[-1]` for å finne det siste elementet i lista
- `min_liste[2:5]` for å hente ut en del-liste som inneholder alle elementene med indeks fra 2 til, men ikke med, indeks 5.
- `min_liste[1:10:2]` for å hente ut en del-liste som med indeks fra 1, til med ikke med 10, med ett hopp på annenhvert element. Det siste tallet bestemmer hvor lange vi skal 'hoppe', så `min_liste[1:10:3]` vil gi hvert tredje element.

Det finnes mange andre nyttige metoder du kan søke opp og sjekke ut dersom du er nysgjerrig, for eksempel `zip`, `enumerate`, `pop`, og `insert`.

5.3 Oppslagsverk

Vi skal nå ta en titt på en annen måte å samle variabler på, som ikke er lister. Oppslagsverk, eller *dictionaries* knytter variabler (eller *elementer*) til nøkler (*keys*) i stedet for en indeks. Dette gjør at plasseringen ikke spiller noen rolle, og det kan være lettere å få oversikt over hvile variabler som representerer hva.

For å lage et oppslagsverk bruker vi krøll-parenteser.

```
min_dict = {}  #et tomt oppslagsverk
```

Si at vi skal samle informasjon om en som heter Petter, som er 32 år gammel og er bosatt i Oslo med postnummer 1181. Med en liste ville vi kanskje gjort noe slikt.

```
person = ['Petter',32,'Oslo',1181]
```

Her må vi selv vite hvilken indeks i lista person som representerer hvilken verdi. Med ett oppslagsverk, kan vi istedet gjøre det slik.

```
person = {'navn': 'Petter','alder':32,'sted':'Oslo','
   postnr':1181}
```

Hvert element presenteres først med en nøkkel (f.eks 'alder') etterfulgt av et kolon, og elementet som tilhører nøkkelen (f.eks 32). Vi henter ut informasjon fra oppslagsverket med klammeparanteser som i en liste, der vi skriver nøkkelen til informasjonen vi vil ha.

```
print(person['navn'])
print(person['alder'])
```

```
Petter
32
```

Dersom vi ønsker å legge til noe nytt i oppslagsverket, skriver vi inn en ny nøkkel, og elementet det skal tilhøre.

```
person['sivilstatus'] = 'gift'
```

Og vi kan endre på et eksisterende element på samme måte som i en liste

```
person['alder'] += 1
```

Vi kan printe ut hele oppslagsverket slik

```
print(person)
```

```
{'navn': 'Petter', 'alder': 33, 'sted': 'Oslo', 'postnr':
   1181, 'sivilstatus': 'gift'}
```

Dersom vi ønsker å få en oversikt over hvilke nøkler, altså hva slags ting vi har i oppslagsverket vårt, kan vi bruke metoden `.keys()`, som viser fram kun nøklene.

```
print(person.keys())
```

```
dict_keys(['navn', 'alder', 'sted', 'postnr', 'sivilstatus
    '])
```

Når vi velger å bruke lister eller når vi velger å bruke oppslagsverk, vil variere med hva slags problem vi prøver å løse, og hva datastrukturen skal brukes til. Det er også individuelt fra programmerer til programmerer hva man fortrekker. Her må vi prøve oss frem, og kanskje teste begge deler for å finne ut av hva som fungerer best for deg i ulike situasjoner.

5.3.1 Lister i oppslagsverk

Det kan være greit å huske på at verdiene i et oppslagsverk ikke må være tall eller tekst, de kan også være andre objekter. For eksempel så kan man ha en liste inne i et oppslagsverk. Dette kan man bruke til å legge til data etterhvert, eller hvis man bare ønsker å ha en liste assosiert med et datasett.

Et eksempel på bruk av dette kan være en handlekurv:

```
bruker = {'navn':'Sofie', 'id':1, 'handlekurv':[]}
print(bruker)
bruker['handlekurv'].append('bananer')
print(bruker)
bruker['handlekurv'].append('appelsiner')
print(bruker)
```

```
{'navn': 'Sofie', 'id': 1, 'handlekurv': []}
{'navn': 'Sofie', 'id': 1, 'handlekurv': ['bananer']}
{'navn': 'Sofie', 'id': 1, 'handlekurv': ['bananer', '
    appelsiner']}
```

Her har vi en enkel handlekurv som tilhører Sofie som er bruker 1. Hun starter med en tom handlekurv, så legger hun til bananer og appelsiner. Hadde vi bare gjort det med en tekststreng så måtte vi tatt og kopiert strenger, lagt til appelsiner og til slutt skrevet inn. Ved å bruke en liste så kan vi smertefritt ha dynamisk innhold i et oppslagsverk.

5.4 Mer om nøkler

I et oppslagsverk er hver nøkkel unik, altså er det ett element knyttet til hver enkel nøkkel. Nøklene kan for eksempel være strenger, heltall eller flyttall.

Man kan for eksempel både skrive

```
prefiks_til_tall = {"milli" : 0.001, "centi" : 0.01, "desi
    " : 0.1}
```

og

```
tall_til_prefiks = {0.001 : "milli", 0.01 : "centi", 0.1 :
     "desi"}
```

og man kan også blande typer man bruker som nøkler (i den grad det kan være hensiktsmessig).

Nøkler kan *ikke* være objekter der innholdet kan endres (*muteable types*); f.eks. kan man ikke bruke en liste eller et oppslagsverk som en nøkkel.

Vi kan prøve, bare for å se hva som skjer. La oss prøve med følgende oppslagsverk:

```
koordinater = {[0, 0] : "a", [0, 1] : "b"}
```

der vi tenker oss at *0, 0* og *0, 1* er koordinater i planet, og *a*, *b* er tilhørende punkter. Prøver vi å kjøre koden her får vi en feilmelding:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Dette kommer først og fremst av et valg utviklerne bak Python har gjort – man *kunne* teoretisk sett hatt lister som nøkler, for man kan sjekke om lister er like eller ikke. Dette er imidlertid en operasjon som kan ta lang tid for lange lister, og man bestemte seg altså tidlig for å bare si at dette rett og slett ikke er mulig.

Litt senere i dette kapittelet skal du lære om noe som heter tupler, som er ganske like lister, men ikke kan endres – og dermed blir vår løsning på hvordan gi koordinater som nøkler. Tupler er nemlig mulig å bruke som nøkler. Vi kommer tilbake til hvordan tupler fungerer – men bare for å vise hvordan syntaksen blir:

```
koordinater = {(0, 0) : "a", (0, 1) : "b"}
```

– og dette er altså kode som vil fungere helt utmerket!

5.4.1 Nøkler som ikke finnes

En feilmelding du helt sikkert vil komme borti når du begynner å jobbe med oppslagsverk, er noe som kalles *KeyError*, som kan se ca slik ut:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'kilo'
```

som vi vil få hvis vi prøver å skrive

```
prefiks_til_tall = {"milli" : 0.001, "centi" : 0.01, "desi
   " : 0.1}
print(prefiks_til_tall["kilo"])
```

Det kommer rett og slett av at vi prøver å skrive ut informasjon tilhørende en nøkkel som ikke er definert. Vi kan bare hente ut den informasjonen vi har lagt inn – her kan vi altså bruke «milli», «centi» og «desi», men vi har ikke definert noe for «kilo».

5.4.2 Nøkler til liste

Noen ganger så ønsker man å finne innholder i et oppslagsverk, eller man ønsker av en annen grunn å finne nøklene. En av de enkleste måtene å jobbe med nøklene til et oppslagsverk er å kaste det til en liste For å kaste nøklene inn i en liste så trenger man bare konvertere til en liste:

```
oppslagsverk = {'nøkkel':1, 'verdi':20, 'hus':54}
nøkkel_liste = list(oppslagsverk)
print(nøkkel_liste)
```

```
['nøkkel', 'verdi', 'hus']
```

Det er viktig å legge merke til at bare nøklene, ikke verdiene, overføres når vi kaster oppslagsverket til en liste.

Vi kommer til å se mer på bruken av dette senere.

5.5 Tupler

En *tuppel* (engelsk *tuple*) er en samling av objekter, slik som lister, men til forskjell fra lister kan man altså ikke endre på innholdet etter at tuppelen er opprettet.

Man oppretter en tuppel ved å bruke vanlige parenteser, slik:

```
1 ukedager = ("mandag", "tirsdag", "onsdag", "torsdag", "
    fredag", "lørdag", "søndag")
```

og vi kan hente ut elementer i tuppelen ved å bruke indekser, som starter på 0 slik som for lister:

```
1 print(ukedager[0])
```

vil gi utskriften

```
mandag
```

Man kan imidlertid ikke endre på elementene etter at tuppelen er opprettet. Hvis vi prøver å skrive

```
1 ukedager[0] = "søndag"
```

(som vil fungere for lister!) vil vi få en feilmelding, som ser omtrent slik ut:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

som rett og slett forteller oss at for en eksisterende tuppel kan vi ikke tilordne et element til en indeks.

Vi kan hente ut deler av en tuppel, som da blir lagret i ett nytt tuppel-objekt, på samme måte som lister:

```
ukedager = ("mandag", "tirsdag", "onsdag", "torsdag", "
   fredag", "lørdag", "søndag")

hverdager = ukedager[:5]
helgedager = ukedager[5:]

print(f"Hverdager: {hverdager}")
print(f"Helgedager: {helgedager}")
```

gir

```
Hverdager: ('mandag', 'tirsdag', 'onsdag', 'torsdag', '
   fredag')
Helgedager: ('lørdag', 'søndag')
```

... og vi kan også kombinere to tupler ved +-operatoren:

```
alle_dager = hverdager + helgedager
print(alle_dager)
```

gir

```
('mandag', 'tirsdag', 'onsdag', 'torsdag', 'fredag', 'lø
   rdag', 'søndag')
```

og `alle_dager` er lik `ukedager` som vi startet med – og det kan forøvrig vi også sjekke:

```
ukedager == alle_dager
```

gir

```
True
```

Vi kan *kaste* en liste til en tuppel og omvendt, ved å bruke `list` og `tuple` (husk at vi bruker () for å opprette en tuppel, og [] for å opprette en liste):

```
ukedager_liste = list(("mandag", "tirsdag", "onsdag", "
   torsdag", "fredag", "lørdag", "søndag"))
print("Type: ", type(ukedager_liste))

```

```
4 ukedager_tuppel = tuple(["mandag", "tirsdag", "onsdag", "
    torsdag", "fredag", "lørdag", "søndag"])
```

gir

```
Type: <class 'list'>
Type: <class 'tuple'>
```

Det er selvfølgelig ingen mening i å kaste et objekt til et annet rett etter at man har opprettet det – men det kan være nyttig i andre sammenhenger, for eksempel hvis man på et tidspunkt vil være sikker på at innholdet i en liste ikke blir endret på et senere tidspunkt i programmet.

5.6 Mengder

En *mengde* (engelsk *set*) er også en samling objekter, men man kan bare inneholde ett element én gang. Mengder kan, som lister, endres etter at man har opprettet dem.

Man oppretter en mengde ved å bruke krøllparenteser – som kan være litt forvirrende siden de også brukes for oppslagsverk! Forskjellen er at man ikke har med kolon, og ramser opp innholdet adskilt ved kommaer, slik som for lister og tupler. Vi kan definere en mengde slik:

```
1 frukttyper = {"banan", "eple", "appelsin"}
```

og hvis vi skriver ut `frukttyper` får vi

```
{'appelsin', 'banan', 'eple'}
```

– merk at elementene kommer i sortert rekkefølge! Dette er tilfelle for alle nyere versjoner av Python (men det finnes også eldre versjoner der det ikke skjer). Vi kan legge til et element ved å bruke `add`:

```
1 frukttyper = {"banan", "eple", "appelsin"}
2 frukttyper.add("mango")
```

og utskriften blir nå

```
{'appelsin', 'banan', 'eple', 'mango'}
```

men hvis vi prøver å legge til «mango» to ganger, vil det altså ikke skje noe mer:

```
frukttyper = {"banan", "eple", "appelsin"}
frukttyper.add("mango")
frukttyper.add("mango")
```

– vi får fortsatt bare:

```
{'appelsin', 'banan', 'eple', 'mango'}
```

Indeksering, som vi er vant med fra lister og tupler, vil derimot *ikke* fungere for mengder, fordi det ikke definert noen sammenheng mellom nummereringen og elementene. Vi kan prøve, for å se hva som skjer:

```
print(frukttyper[0])
```

og da vil vi få en feilmelding som ser omtrent slik ut:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

som kanskje er litt kryptisk, men den sier at `set` ikke er *subscriptable*, som er en samlebetegnelse for objekter der man kan hente ut elementer ved indekser. Vi sier at mengder er en *uordnet datastruktur*.

Vi kan sjekke om et element er i mengden vår ved å bruke `in`-operatoren , altså slik:

```
print("mango" in frukttyper)
print("ananas" in frukttyper)
```

som gir henholdsvis

```
True
False
```

Man bruker mengder dersom man jobber med objekter der man ikke trenger å ta med elementer hvis de allerede finnes. Vi kan kaste en liste (eller en tuppel) til en mengde ved å bruke `set`:

```
frukttyper = set(["banan", "eple", "banan", "appelsin", "
   eple"])
```

```
print(frukttyper)
```

som gir

```
{'appelsin', 'banan', 'eple'}
```

... og motsatt:

```
frukttyper = list({"banan", "eple", "appelsin"})
print(frukttyper)
```

som gir oss

```
['appelsin', 'banan', 'eple']
```

En oversikt over alle tilgjengelige funksjoner for mengder kan man f.eks. finne på https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset.

6 Strenger

Strenger er en enkel men ofte brukt datatype. Men siden det er mye mer man kan gjøre med strenger enn det som er åpenbart så skal vi gå litt mer inn på dette.

6.1 Strenger som lister

På mange måter er en streng datatype litt som en liste av bokstaver og tegn. Som et resultat av dette er det mange av tingene vi kan gjøre med lister som også kan gjøres med strenger.

Som vi så på i forrige kapittel så kan man dele opp lister med `liste[2:4]` for å få en delliste. Vi kan gjøre de samme operasjonene på strenger for å få deler av en streng. VI kan også sjekke om et symbol er i strengen og telle hvor mange det er.

```
streng = 'Strenger kan behandles som lister!'
print(streng[0:3]) #skriv ut de første 3 bokstavene
print(streng[0::2]) #skriv ut annenhver bokstav
print('e' in streng)
print(streng.count('e'))
```

```
Str
Srne a eade o itr
True
5
```

Derimot så er det viktig å huske på at mens mange av operasjonene som fungere på lister kan brukes på strenger er det ikke alle. Man kan for eksempel ikke endre deler av en streng, og man kan heller ikke sortere eller reversere den.

```
streng = 'Men ikke alt som fungerer på lister går for
   strenger'
streng[3] = '_'
streng.sort()
streng.reverse()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'sort'
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'reverse'
```

Hvis man har lyst til å endre på deler av en streng så må man dele den opp, bytte det man ønsker, for så å sette den sammen igjen.

```
streng = 'Strengen kan man ikke endre!'
del_1 = streng[0:17]
del_2 = streng[21:]
ny_del = 'definitivt'
ny_streng = del_1 + ny_del + del_2
print(ny_streng)
```

```
Strengen kan man definitivt endre!
```

6.2 Strengemetoder

Noen ganger kan det være nyttig å manipulere, endre på, eller sjekke tekstrenger når vi programmerer, særlig når vi tar inn input fra en bruker, eller leser inn data

fra en fil (dette skal vi lære mer om senere). Vi skal nå se på noen av metodene vi har for å behandle tekst i Python.

La oss ta utgangspunkt i en kort tekst.

```
tekst = 'Hei på deg, jeg heter Sofie! Ha en fin dag.'
```

Vi kan gjøre alle bokstavene i en tekst til små eller store, ved hjelp av metodene `upper()` og `lower()`.

```
print(tekst.upper())
print(tekst.lower())
```

```
HEI PÅ DEG, JEG HETER SOFIE! HA EN FIN DAG.
hei på deg, jeg heter sofie! ha en fin dag.
```

Når vi spør en bruker om et svar, og skal sjekke det med en betingelse, har vi sett at det kan bli vanskelig å få brukeren til å svare akkurat det vi forventer. Om vi lurer på om brukeren har svart 'ja', kan det være lurt å også aksepter f.eks 'Ja' og 'JA'. Tidligere har vi gjort dette ved å utvide if-testen.

```
if svar == 'ja' or svar == 'Ja' or svar == 'JA':
    #gjør noe
```

Ved å bruke `lower()` kan vi slippe dette, ved å først gjøre om svaret til kun små bokstaver.

```
if svar.lower() == 'ja':
    #gjør noe
```

Dersom vi ønsker å spørre brukeren om hva de heter og hvor gamle de er, har vi vanligvis gjort dette i to operasjoner.

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

Med strengemetode `split()` kan vi gjøre dette på samme linje. Den splitter opp en streng in en liste med flere strenger. La oss ta en titt.

```
print(tekst.split())
```

```
['Hei', 'på', 'deg,', 'jeg', 'heter', 'Sofie!', 'Ha', 'en
   ', 'fin', 'dag.']
```

Som vi kan se, splitter den opp strengen der det er mellomrom. Så dersom vi ber om navn og alder på samme linje, kan vi hente det ut som elemeter fra en liste.

```
svar = input('Skriv inn navn og alder:')
svar = svar.split() # gjør til formen ['navn','alder']
navn = svar[0]
alder = svar[1]
```

split() deler i utgangspunktet opp ved mellomrom, men vi kan også spesifisere hva vi vil at det skal dele ved, f.eks ved komma:

```
print(tekst.split(',')
```

```
['Hei på deg', ' jeg heter Sofie! Ha en fin dag.']
```

– eller ved både komma og mellomrom:

```
print(tekst.split(', ')
```

```
['Hei på deg', 'jeg heter Sofie! Ha en fin dag.']
```

Å kunne splitte strenger er særlig nyttig når vi har informasjon ramset opp og adskilt ved hjelp av et bestemt tegn (her komma); da kan vi hente ut informasjonen fra en streng og legge den over i en liste:

```
farger = "rød, orange, gul, grønn, blå, lilla"

print(farger.split(", "))
```

```
['rød', 'orange', 'gul', 'grønn', 'blå', 'lilla']
```

Vi kommer til å bruke dette mye når vi skal jobbe med *csv-filer*; csv står for «comma-separated values». Vanligvis består slike filer av en tenkt «tabell» der kolonnene er avgrenset av komma-tegn og radene ved linjeskift.

7 Løkker

Løkker brukes for å gjenta en prosess mange ganger. Dette høres kanskje ikke så nyttig ut, men er en 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 engelsk. I bunn og grunn er de to veldig like, og går begge ut på å gjenta en prosess mange ganger.

7.1 Historie: Shampoo-algoritmen

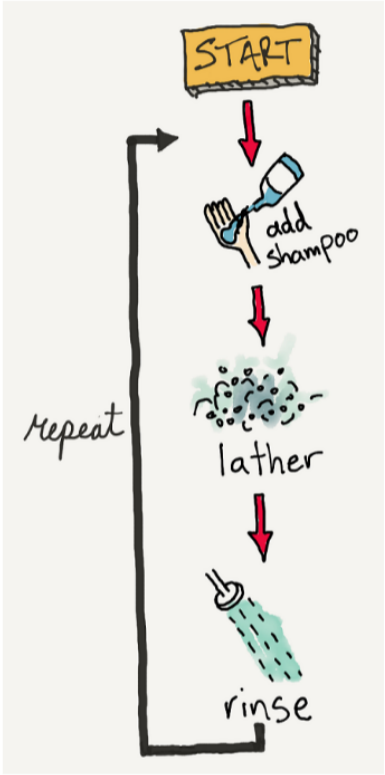
Sjampoalgoritmen stammer fra instrukser som man finner på mange sjampo-flasker. På engelsk er disse ofte:

- 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

7.2 For-løkker

Vi ønsker å lage et program som skriver ut en hilsen til hver ansatt i en bedrift, og vi har lagret de ansattes navn i en liste. Da kan vi for eksempel skrive slik:

```
ansatte = ['Alma', 'Filip', 'Sofia', 'Mohammad', '
   Alexander', 'Sara']
print(f'Hei {ansatte[0]}')
print(f'Hei {ansatte[1]}')
print(f'Hei {ansatte[2]}')
print(f'Hei {ansatte[3]}')
...
```

Men dette blir fort slitsomt og vi ser at vi gjentar veldig lik kode mange ganger. Det må da finnes en bedre måte og gjenta samme koden for hvert element i en liste?

For løkken er laget for nettopp slike problemer. Ved å bruke en for-løkke kan vi korte ned koden på denne måten:

```
ansatte = ['Alma', 'Filip', 'Sofia', 'Mohammad', '
   Alexander', 'Sara']
for ansatt in ansatte:
    print(f'Hei {ansatt}')
```

Python syntaksen for å opprette for løkker er ganske lik hvordan vi ville sagt det som "menneske språk". Hvis vi skulle gitt et menneske instrukser om å hilse på alle ansatte i et rom kunne vi sagt noe lignende som: "for hver ansatt i dette rommet: Hils på dem".

Koden over sier omtrent dette. Vi ber Python om å "for hver ansatt i lista over ansatte: skriv ut hilsen til den ansatte". En for-løkke er altså en måte vi kan gjenta samme biten av kode for hvert element i en liste.

La oss gå igjennom hver bit av Python-syntaksen for første linje i `for`-løkka over.

Nøkkelordet `for` For å starte en `for`-løkke bruker du alltid nøkkelordet `for`.

Løkkevariabelen `ansatt` `ansatt` er navnet på løkkevariabelen. Dette er variabelen som «husker på» hvilket liste-element vi arbeider med for øyeblikket. Første runde i løkka vil `ansatt` ha verdien `'Alma'` (det første elementet i lista `ansatte`. Andre runde i løkka får `ansatt` verdien `'Filip'` (det andre elementet i lista). Slik fortsetter løkka helt til alle elementene i lista er gått igjennom. Du er fri til å gi løkkevariabelen hvilket som helst navn, men det er fornuftig å gi et navn som henger sammen med hva variabelen inneholder.

I dette tilfelle er går løkka igjennom en liste av ansatte så løkkevariabelen er en ansatt. Derfor er det naturlig å kalle den `ansatt`

Nøkkelordet `in` `in` er det andre nøkkelordet du trenger for å definere en `for`-løkke.

Listevariabelen `ansatte` . Denne variabelen representerer hvilken liste [1] vi ønsker å løkke igjennom. I dette tilfelle er det altså lista `ansatte` som inneholder navnene på de ansatte i en liten bedrift.

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 med to piler som peker hver sin vei like over Caps Lock, på venstre side av tastaturet.

Hvis vi gjør en tabbe med innrykkene, så får vi samme feil som vi nevnte i Betingelser 4.1.3, nemlig `IndentationError`. Disse tabblene kan gi uønskede resultater self om koden kjører.

7.3 Løkke over tallrekker med for-løkker

Det hender ofte at det vi ønsker å løkke over er en tallrekke. For eksempel hvis vi vil løkke over alle år mellom 2020 og 2030 eller over alle aldere fra 0 til 30 år.

For å løkke over alle tall fra 0 til 10 kan vi for eksempel bruke en liste:

```
1 for tall in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
2     print(tall)
```

```
1
2
...
```

[1] Strengt tatt kan du løkke over flere typer variable enn bare *lister* (det holder at det du løkker over er en *samling*). Men for nå holder det å tenke på at det er lister vi løkker igjennom.

```
10
```

Men det kan fort bli slitsomt å skrive ut en lang liste av tall og det er lett å skrive feil. Tenk deg om vi skal løkke over alle tall fra 0 til 100! Det blir en alt for lang liste å skrive ut!

Heldigvis er dette et så vanlig problem at det finnes en egen Python funksjon for å generere en tallrekke som vi kan løkke over. Denne funksjonen heter `range()`, hvilket er engelsk for "verdimengde"

For å lage en løkke over tallrekka 1, 2, ..., 10 ved hjelp av `range`, så skriver vi

```
1 for tall in range(1, 11):
2     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.

7.3.1 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):
```

```
    total += tall

print(total)
```

```
55
```

Her er det to ting å merke seg spesielt. Først så bruker vi `+=` her, altså oppdaterer 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.

7.3.2 Finkontroll på tallrekka

I alle eksemplene så langt har vi brukt `range` med to tall, nemlig `range(fra, til)`. Vi kan 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.

7.3.3 Nøkler, lister og for-løkker

Hvis man ønsker å finne verdiene til hver nøkkel i et oppslagsverk, for eksempel for å legge de sammen eller for å behandle de i en egen liste, så kan man bruke en for-løkke over nøkkel-listen:

```
verdi_liste = []
for nøkkel i nøkkel_liste:
     verdi_liste.append(oppslagsverk[nøkkel])
print(verdi_liste)
```

```
[1, 20, 54]
```

Legg merke til at siden vi har nøkklene i en liste, så kan vi bruke verdiene i listen som nøkler i oppslagsverket. Dette lar oss få ut alle verdiene i samme rekkefølge som nøklene. Men andre ord, hvis vi sorterte nøklene så ville vi også få en sortert verdiliste.

Hvis man bare ønsker å løkke gjennom nøklene uten å sortere eller bry deg om rekkefølgen så kan man gjøre det samme på en annen måte:

```
for nøkkel in oppslagsverk.keys():
    verdi_liste.append(oppslagsverk[nøkkel])
```

Hvilken man bruker avhenger av hva man ønsker å gjøre med nøklene, og litt personlig preferanse.

7.4 Å gjenta kode n ganger

For å gjenta en kode n ganger 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, og det er ikke nødvendig å bruke.

7.4.1 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. Vi kan imidlertid bruke en `for`-løkke til å gjøre det kjapt og effektivt.

Hvis vi vil ha en norsk versjon kan vi gå til sangen om fiskebolla som lengter etter havet – slik:

```
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!
```

7.5 While-løkker – til noe er sant

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 derimot, er 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
```

```
    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.

7.5.1 Eksempel: Høyrentekonto

Når vi så på `for`-løkker så vi på et eksempel der vi regnet ut hvor mye penger som var på en sparekonto for hvert år som gikk, når vi 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
```

```
4
5  while penger < 20000:
6      penger *= rente
7      år += 1
8
9  print(f'Du må vente i {år} år.')
10 print(f'Det er da {penger} på konto.')
```

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

7.5.2 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.

```
1  svar = input("Hva heter hovedstaden i Portugal? ")
2
3  while svar != 'Lisboa':
4      svar = input("Nei, det var feil. Prøv igjen. ")
5
6  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!
```

7.5.3 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. Det er imidlertid sånn at alle problemer som kan løses med den ene typen kan også løses med den andre. I praksis handler det altså mest om hvordan man liker å tenke og personlig preferanse.

7.5.4 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 opp 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.

7.6 Avansert 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 Python-program for å finne ut dette.

7.6.1 Høyrentekonto med for-løkker

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 hvert 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 tillegg 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.

7.7 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

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

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

```
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 visualiserer hvordan pengemengden øker mer og mer for hvert år.

7.8 Avansert eksempel: 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. Likevel 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.

7.8.1 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*.

7.8.2 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 og vise *én* mulig måte å jobbe seg igjennom dette problemet. Men det er fullt mulig å gå frem på alternative måter.

Et naturlig første steg er å ignorere tilleggsreglene våre fullstendig, og 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 glemme 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 tillegg 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»:

```python
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 å tro at vi er ferdig, men det er en liten justering til 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, som sjekker om tallet er delelig med begge. Denne testen må legges før de andre, prøv gjerne å forstå hvorfor. Med dette lagt inn blir hele programmet vårt

```python
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 laget 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 er det bare kreativiteten som setter grensene. Hva med å si at alle tall i ti-gangen skal hoppes helt over?

7.8.3 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.

8 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 `sort` 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 funksjon 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 `sort`-funksjonen for å sortere en liste.

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 finne ut lengden til teksten `'Ansatte'` skriver vi `len('Ansatte')`. Vi sier da at `'Ansatte'` er input til lengdefunksjonen, eller *argumentet*.

8.1 Definere egne funksjoner

I tillegg 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 kutt(argument):
2     return argument.lower().split()
```

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 `kutt`. 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 *argument*. Til slutt trenger vi et kolon, slik som i løkker og tester. Merk at `kutt` og `argument` er valgfrie navn vi har valgt helt selv.

Etter denne linjen kommer innholdet i selve funksjonen, og her trenger vi et inn-

rykk. 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 splitter den ukjente variabelen *argument*, 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.

Når vi kjører kodesnutten over, så *definerer* vi funksjonen, det betyr at etter disse kodelinjene er kjørt, så eksisterer `kutt` 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 `kutt('Python Er Kult')`, og da kjøres kodelinjene inne i funksjonen med `argument ='Python Er Kult'`, la oss teste dette selv:

```
print(kutt('Python Er Kult'))
```

```
['python', 'er', 'kult']
```

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

```
print(kutt('Funksjoner gjør ting raskt'))
print(kutt('og RYDDIG!'))
print(kutt('Men viktigst av ALT'))
print(kutt('Det gjør ting enkelt'))
```

```
['funksjoner', 'gjør', 'ting', 'raskt']
['og', 'ryddig!']
['men', 'viktigst', 'av', 'alt']
['det', 'gjør', 'ting', 'enkelt']
```

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

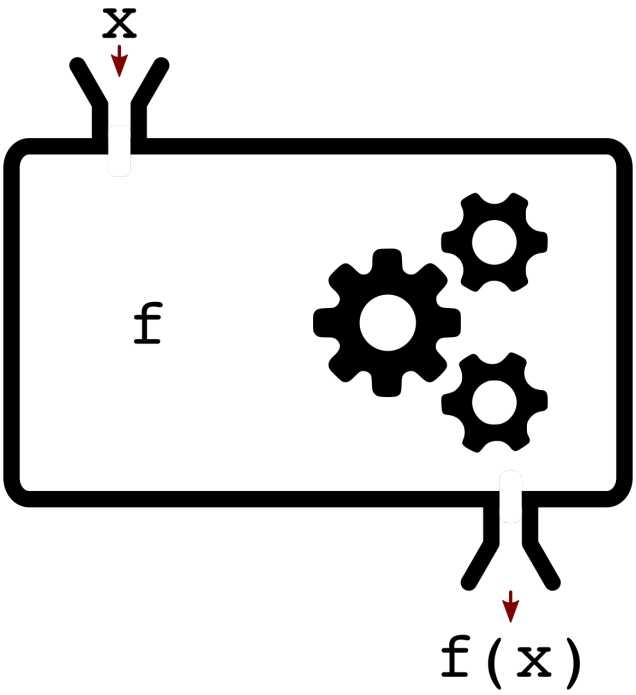
```
resultat = kutt('Fuksjoner gjør ting raskt')
```

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

8.2 Analogier for funksjoner

Funksjoner kan være et litt vanskelig konsept å forstå, så det er viktig å bruke tid på og reflektere over hva funksjoner egentlig *er*.

En 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.

8.3 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 regner ut gjennomsnittet av to tall, da kan vi definere den slik.

```
def gjennomsnitt(a, b):
    return (a + b) / 2
```

Tilsvarende kan vi fint lage en funksjon som returnerer mer enn én ting, da legger vi dem bare etter hverandre og deler dem med komma. For eksempel så kan vi utvide gjennomsnitt funksjonen vår til å returnere forskjellen mellom de to tallene i tilegg.

```
def gjennomsnitt(a, b):
    if a > b:
        forskjell = a - b
        snitt = (a + b) / 2
    elif b > a:
        forskjell = b - a
        snitt = (a + b) / 2
    else: #tallene er like
        forskjell = a-b # skal bli null
        snitt = (a+b) / 2 # skal være a==b
    return snitt, forskjell

a = 9
b = 5
snitt, forskjell = gjennomsnitt(9, 5)
print(f'Gjennomsnittet til {a} og {b} er {snitt}, og
   differensen er {forskjell}')
```

```
Gjennomsnittet til 9 og 5 er 7, og differensen er 4
```

8.4 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 over at funksjoner i seg selv faktisk *er* variabler. Som andre variabler har de et navn, og et innhold. I dette tilfellet derimot, har ikke variabelen tekst eller tall som innhold, men *kode*, og denne koden kjøres når man bruker funksjonen.

Det er også verdt å merke seg at variabler vi definerer inne i en funksjon kun kan brukes inne i den funksjonen. Det kan kanskje virke litt klønete når du ikke er vant med det, men det har en del fordeler. La oss si vi har flere funksjoner. Disse kan brukes om hverandre, men gjør forskjellige ting. Det kan hende man ønsker å bruke samme navn på en variabel inne i en funksjon som man har brukt i filen et eller annet sted. Isteden fro å måtte ha forskjellige navn på alle variabler, så kan vi gjenbruke navn inne i funksjonene våre.

Samtidig så kan man bruke variabler fra hele dokumentet inne i funksjoner hvis man *ikke* definerer eller endrer på de ikke i funksjonen. La oss se på et eksempel:

```
1  def funksjon_1():
2      var = 5
3      print(var)
4
5  def funksjon_2():
6      print(var)
7
8  var = "en global variabel"
9  funksjon_1()
10 funksjon_2()
```

```
5
en global variabel
```

Her har vi to funksjoner. Før vi bruker funksjonene definerer vi en variabel `var`, dette er en *global variabel*, det vil si at den gjelder for hele Python filen. Deretter kjører vi to funksjoner. Den første, `funksjon_1()`, definerer også en variabel som

heter `var` før vi printer ut variabelen. Dersom vi ikke hadde definert global `var` før vi brukte funksjonen så hadde denne funksjonen skrevet ut det samme. Vi har også `funksjon_2()`. Denne skriver bare ut den *globale variabelen*, det vil si at hvis vi ikke hadde definert `var` i filen utenfor en funksjon så hadde denne funksjonen krasjet.

8.5 Funksjoner i programmering

Eksemplene vi har sett på så langt har vært ganske enkle funksjoner. I programmering kan funksjoner derimot gjøre nesten hva som helst. Dette er fordi de kan inneholde hva som helst av kode og det betyr at de kan gjøre alt som vi ellers gjør i programmene våre. Et helt lite program kan være kun en funksjon i et større program.

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

```
def hils_på(navn):
    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!
```

8.6 Avansert eksempel: Tekstanalyse

For et litt større og mer omfattende eksempel så skal vi se på litt tekstbehandling. Det er mange måter man kan behandle og jobbe med tekst i Python, men vi skal bruke noen enkle metoder for å få noe nyttig ut av tekst. I første omgang vil det være hvor mange ganger ord dukker opp i en tekst og hvor mange ord det er i setninger. Fra dette kan man for eksempel lage en ordsky eller se på "kompleksiteten" til teksten basert på setnings lengde.

Programmere tekstbehandling

Vi vil programmere tekstbehandleren vår en funksjon av gangen. Den første vi skal starte med er orddeleren. Denne funksjonen vil ta en streng som input og dele den opp i enkeltord. For å gjøre dette bruker vi `split()` funksjonen. Men siden denne også kan gi oss "tomme" ord så må vi ha litt ekstra logikk for å bli kvitt disse:

```
def ord_deler(streng):
    '''
        Tar inn en streng og deler den i enkelt ord basert
        på mellomrom

        Returnerer en liste med ord
    '''
    ord_liste = streng.split()
    ny_liste = []
```

```
    for ord in ord_liste:
        if ord or ord == 0:
            ny_liste.append(ord)
    return ny_liste
```

Den ekstra logikken kommer i form av en for-løkke med en if-setning. Siden en tom streng og en liste er såkalte *Falsy* verdier, det vil si at hvis du tester på de så gir de `False` som svar, så bruker vi det for å se om det er noe inhold. Men siden 0 også er en *Falsy* verdi så legger vi inn en ekstra sjekk for det. Dersom det er en verdi vi ønsker så legger vi det inn i en ny liste og returnerer denne.

Legg merke til at det er lagt inn en *kommentar* øverst i funksjonen. Denne er her for å gi en rask forklaring av hva funksjonen tar som input, hva den gjør og hva den returnerer. Dette gjør det lettere å komme tilbake senere for å bruke den, samtidig som det gjør det lettere for andre å se hva funksjonen brukes til.

Kommentarer kan skrives på to måter:

```
# en linjekommentar starter med en
# # og varer ut lengden på linjen
'''
Mens en blokkommentar er omsluttet av trippel fnutter
og kan være på flere linjer
'''
```

Det å bare ha ord delt opp i en liste er i seg selv ikke så nyttig, så la oss gjøre noe med listen. Vi lager en funksjon som tar listen og teller opp ordene den finner og returnerer det som et oppslagsverk. Dette lar oss se på antallet mye lettere enn hva en liste ville:

```
def ord_teller(input_liste):
    '''
        Tar inn en liste med ord og teller hvor mange
        ganger hvert ord dukker opp

        Returnerer en dict med ordene og antall ganger
    '''
    ord_teller = {}
    for ord in input_liste:
        ord_teller[ord.lower()] = ord_teller.get(ord.lower
           (),0) +1
    return ord_teller
```

Som i forrige bit benytter vi oss også her av en for-løkke. I denne løkken sjekker vi hvert element i input listen og legger det til i et oppslagverk hvis det ikke ligger det allerede. Men dersom det allerede er der så teller vi opp en verdi. På denne måten kan vi telle hvor mange ganger et ord dukker opp i en tekst. Det er verdt å merke at vi har lagt til en `lower()` før vi legger det inn i oppslagsverket. Dette er lagt inn som en ekstra sikkerhet mot at vi kan få samme ord flere ganger hvis det er skrevet med store eller små bokstaver, f.eks. "er", "Er" og "ER".

Den neste funksjonen vi skal legge til er for å dele opp setninger. Dette kan kanskje virke litt unødvendig når vi allerede har funksjoner som kan dele opp i ord, men det kan være nyttig for å analysere en tekst. For eksemper er setningslengden ofte brukt som en pekepinn på hvor vanskelig en tekst er. Denne funksjonen kommer til å være litt lenger enn de vi har skrevet så langt, så her er det viktig å holde tunga rett i munnen:

```
1  def setning_deler(streng):
2      '''
3          Tar inn en streng og deler den i setninger basert
               på
4          end of line tegnene (!.?)
5
6          Returnerer en liste med setninger
7          Den returnerte listen er kun med små bokstaver
8      '''
9      midlertidig_liste = list()
10     setnings_liste = []
11     midlertidig_liste.append(streng.split('.'))
12     setnings_liste = midlertidig_liste.copy()
13     midlertidig_liste.clear()
14     for setning in setnings_liste:
15         for medlem in setning:
16             midlertidig_liste.append(medlem.split('!'))
17     setnings_liste = midlertidig_liste.copy()
18     midlertidig_liste.clear()
19     for setning in setnings_liste:
20         for medlem in setning:
21             midlertidig_liste.append(medlem.split('?'))
22     setnings_liste = midlertidig_liste.copy()
23     midlertidig_liste.clear()
```

```
24      endelig_liste=list()
25      for setning in setnings_liste:
26          for medlem in setning:
27              if medlem:
28                  endelig_liste.append(medlem.lower())
29      return endelig_liste
```

Her starter vi med å ta inn en streng som inneholder en tekst. Mens de fleste strengene vi har jobbet med så langt har vært korte, så kan strengen også inneholde en hel bok for å la oss analysere den. Denne strengen blir så *splittet* med funksjonen `split('.')`. Dette gjør at isteden for å dele på hvert mellomrom så deler vi på hvert punktum.

Vi tar så å *kopierer* listen. Dette er viktig siden hvis man bare bruker '=' så vil man ikke lage en ny liste, kun en *referanse*. Dette ville gjort at alle operasjoner vi gjorde på den ene listen også ville blitt gjort på den andre listen. Dette er fordi så vidt datamaskinen vet, er det den samme listen med to forskjellige navn, litt som hvordan «København» og «Copenhagen» begge refererer til samme by, men med to forskjellige navn.

Vi fortsetter deretter med å bruke `clear()` på den midlertidige listen. Dette tømmer listen for alt innhold og gjør at vi unngår at det blir noe kluss med dataene vi jobber med. Deretter skal vi gjøre den samme prosessen to ganger, en gang med «!» og en gang med «?». *Om du har lyst på en ekstra utfordring: prøv å skille ut dette i en egen funksjon så du bare trenger å gjøre et kall med det symbolet du ønsker å splitte på.* Vi lager en for løkke som løkker gjennom setningslisten vår, splitter på "!" og på "?", for så å kopiere den nå ryddigere listen over den gamle ordlisten og tømme den midlertidige listen.

Det siste vi skal gjøre før vi er ferdig med denne funksjonen er å ta listen vi har med setningene våre. Dette er en liste med lister, men hver av listene inneholder bare en streng. For å gjøre dette til et litt med vennlig format bruker vi en for-løkke for å gå gjennom listen, så tar vi samme sjekken vi brukte tidligere for å fjerne tomme lister. Tilslutt får vi da en liste med alle setningene i strengen vi startet med og returnerer den.

Nå som vi har setningene delt opp, så må vi jo kunne gjøre noe nyttig med dem. For eksempel, så kan vi sjekke lengden til setningene. I første omgang lager vi en funksjon som tar inn en streng for så å sjekke hvor mange ord den har:

```
1  def setnings_lengde(input_streng):
2      '''
```

```
        Tar inn en streng og returnerer hvor mange ord som
            er i setningen
        Krever at input strengen kun er en setning

        Returnerer en integer lengde
    '''
    ord_liste = ord_deler(input_streng)
    return len(ord_liste)
```

Her får vi et godt eksempel på *nesting* av funksjoner, eller funksjoner inne i funksjoner. Det vil si at vi bruker en av våre egne funksjoner inne i en annen funksjon. Vi kan kalle på alle funksjoner i filen inne i alle andre funksjoner. Her bruker vi orddelingsfunksjonen vi lagde tidligere for å dele opp setningen i ord, for så å gi oss lengden av listen. Dette lar oss lett finne ut hvor mange ord en setning inneholder.

Til slutt bruker vi de funksjonene vi har laget for å regne ut den gjennomsnittlige setningslengden til teksten:

```
def setnings_lengde_snitt(input_liste):
    '''
        Tar inn en liste av setninger og regner
        ut snitt lengden på de

        Returnerer en float lengde
    '''
    lengder = list()
    total_lengde = 0
    for setning in input_liste:
        lengder.append(setnings_lengde(setning))
    for lengde in lengder:
        total_lengde += int(lengde)
    snitt = total_lengde/len(lengder)
    return snitt
```

Her tar vi inn en liste med setninger (f.eks. generert av `setnings_deler()` funksjonen) og regner snittet. For å gjøre dette tar vi først å *initialiserer* listen `lengder` og tallet `total_lengde`. Deretter har vi en for-løkke for å ta setningslengden for hver setning i listen og legger dette inn i en ny liste. Denne listen blitt så brukt i en annen for-løkke for å legge sammen alle lengdene til den *totale lengden* på teksten. Denne totale lengden blir så delt på lengden til listen `lengder`, dvs. antallet setninger, dette gir oss et gjennomsnitt av lengden til setningene i teksten.

9 Filhåndtering

Hittil har du lært mange grunnleggende konsepter, og dette er viktige byggesteiner som du må ta med deg videre. I dette kapittelet skal vi lære teknikker som forhåpentligvis gjør det mulig for deg å kombinere programmering med praktiske anvendelser – vi skal lære å jobbe med filer.

Det er mye man kan gjøre med programmering for å løse problemer der og da, men det hender man vil lagre ting til senere bruk, eller man ønsker å lese noe fra en fil eller et annet program. Å skrive det inn manuelt hver gang blir veldig tungvint, men heldigvis kan man jobbe direkte med filer i Python.

9.1 Filer i Python

Når man jobber med filer i programmering er det ikke som å åpne et Word dokument eller en fil på PCen. Du ser ingenting når filen er åpen, og man kan ofte ikke jobbe direkte med den. Det man kan gjøre derimot er å *lese* innhold fra filen for å jobbe med det, for så å *skrive* til filen etter man er ferdig. På samme måte som ingenting er lagret i Word før du klikker på lagre så er ingenting av det du gjør i programmet lagret før det skrives til en fil. Det er derfor viktig å skrive ting man ønsker å lagre til filer før man lukker programmet.

9.2 Lese fra filer

På samme måte som man ikke klarer å lese innholdet i en bok uten å åpne coveret kan man ikke lese innholdet i en fil uten å åpne den først. Det man må være ekstra obs på når man jobber med datamaskiner derimot er at hvis du ikke ber dem om å lukke boken etterpå så vil de la den være åpen i evig tid, eller i hverfall til programmet avsluttes. Man kan da ende opp med at datamaskinen har en berg med åpne filer samtidig, hvilket kan gjøre at datamaskinen blir treg. Derfor må man alltid huske at man kan ikke bare fortelle en datamaskin at den må åpne filen, men også lukke den igjen.

9.2.1 Å åpne en fil

For å åpne filer i Python brukes `open()` funksjonen. Det er hovedsakelig to måter å bruker den på. Den første er enkel, men du må passe på å lukke filen selv etterpå:

```
f = open(fil)
fil_innhold = f.read()
f.close()
```

En annen, mer robust måte som automatisk lukker filen når du er ferdig med den en `with` funksjonen:

```
with open(fil) as f:
    fil_innhold = f.read()
```

Disse to kodesnuttene gjør nøyaktig det samme, så hvordan du ønsker å åpne filen er opp til deg. Som du kanskje ser så er det en `read()` funksjon som er "hektet på" `f` i kodesnuttene. Denne funksjonen leser innholdet i den åpnede filen som en lang tekst streng. Denne vil nødvendigvis måtte viderebehandles avhengig av bruk. Man kan også bruke filer uten å lese den inn som tekst.

Man kan også lese enkelt linjer isteden for hele dokumentet med `readline()` isteden for `read`.

En feilmelding som fort kan dukke opp er en formaterings feil. Dette kan forekomme av at filen din bruker et annet *tegnsett* enn det Python prøver å lese. Python prøver å lese det formatet som er standard på datamaskinen din. For å fikse dette så må vi legge til et definert tegnsett.

```
with open(fil, encoding="utf-8", errors="replace") as f:
    fil_innhold = f.read()
```

UTF-8 er et standard tegnsett, *encoding* på engelsk, som brukes på det aller meste av moderne filer. Men det skader skjeldent å definere hva slags tegnsett filene skal bruke, spesielt hvis koden skal kunne kjøre på andre datamaskiner enn din egen. Det er også lagt til en ekstra bit i `open()` funksjonen, `errors="replace"`. Denne gjør at selv hvis Python ikke klarer å åpne filen som den skal, så vil innholdet fortsatt bli spyttet ut med de symbolene Python ikke klarer å lese blir erstattet. Man må derimot være forsiktig med det, siden man risikerer å få masse søppeldata hvis det er en feil i filen.

9.2.2 Håndtering av filinnhold

Når vi har fått åpnet filen vår, og lagret dem som en lang tekst, kan vi hente ut informasjonen vi ønsker fra dem. Hvordan dette gjøres kan være avhengig av hvordan filen ser ut. Et eksempel kan være en tekstfil `middager.txt` som består av to kolonner, en kolonne med ukedager, og en med hva man skal spise

```
mandag      laks og ovnsbakte grønnsaker
tirsdag     gulrotsuppe
onsdag      kyllingwok
torsdag     mossaka
fredag      spansk gazpacho
```

Hvis vi leser inn denne med

```
with open('middager.txt') as f:
    fil_innhold = f.read()
```

Vil vi ha en strengevariabel `fil_innhold`, som inneholder hele teksten. Si at vi ønsker å få dette inn i et oppslagsverk, som gjør at vi kan slå opp på en dag og få ut dagens middag.

Vi husker at vi kan bruke `split()` til å dele opp en streng. Først vil vi dele opp strengen der det er linjeskille, det gjør vi ved symbolet `'\n'`, som betyr ny linje. Deretter kan vi splitte opp hver linje der det er et større mellomrom, ved å bruke `'  '` (tab/fire mellomrom).

```
with open('middagsforslag.txt', encoding='utf-8') as f:
    fil_innhold = f.read()

middager = {}  # tomt oppslagsverk
linjer = fil_innhold.split('\n')
linjer = list(filter(None, linjer))  # fjern eventuelle
   tomme strenger
for linje in linjer:
    (dag,middag) = linje.split('    ')
    middager[dag] = middag.strip()

print(middager)
```

```
{'mandag': 'laks og ovnsbakte grønnsaker', 'tirsdag': '
   gulrotsuppe', 'onsdag': 'kyllingwok', 'torsdag': '
   mossaka', 'fredag': 'spansk gazpacho'}
```

Dette er en måte å håndtere en fil på, men hva som er den riktige fremgangsmåten vil variere med hvordan filen ser ut. Her må vi justere oss, og bruke det vi har lært om strengemetoder til å hente ut den informasjonen vi vil ha.

9.2.3 Lese fra CSV-filer

Det kan være frustrerende at ulike filer bruker ulike måter å lagre data på, med linjeskift, antall mellomrom osv. Heldigvis finnes det en standard, som er mye brukt for filer som inneholder større mengder data. Dette formatet er CSV, *comma, separated variables.*. Her lagres informasjon som hører sammen på en linje, med komma som separerer hvert dataelement. Ofte er den øverste linja en beskrivelse av hva de ulike posisjonene representerer. Her er et utdrag av et eksempel `ansatte.csv` som beskriver navn, alder, inntekt og lokasjon til ansatte i en bedrift.

```
navn,alder,inntekt,kontor
Marie,30,403279,Stavanger
Fredrik,48,310765,Stavanger
Arne,71,515931,Stavanger
Gunnar,28,436807,Oslo
Martin,41,539709,Oslo
Turid,41,491790,Stavanger
Laila,34,495055,Trondheim
Anna,51,344160,Oslo
Helge,46,451056,Stavanger
Liv,51,676788,Trondheim
Emma,50,647031,Oslo
Heidi,37,607718,Stavanger
Elisabeth,26,401886,Stavanger
```

Beleilig nok, finnes det et eget Python-bibliotek for å behandle denne typen filer. `csv`-biblioteket er laget spesielt for å behandle CSV-filer. For å bruke `csv`-biblioteket må vi først importere det.

```
1 import csv
```

Nå som vi har biblioteket importert så må vi jo gjøre noe med det. I første omgang kan vi åpne filen vi nevnte tidligere og lese inn de ansatte dataene. For å gjøre dette bruker vi `csv.reader()`. Dette er CSV-bibliotekets funksjon for å lese CSV-filer som et csv objekt. Dette objektet fungerer på mange måter på som en liste av lister. Det som er viktig å huske på når man bruker `csv.reader()` er å definere `delimiter`. Dette er det tegnet som brukes for å dele opp colonner. Selv om mange CSV-filer bruker komma for å dele opp, så er det ikke alle.

```
1
2 with open('.\\ansatte.csv', newline='') as csvfil:
3     csvleser = csv.reader(csvfil, delimiter=',')
4     for rad in csvleser:
5         print(f'{rad[0]:10} {rad[1]:10} {rad[2]:10} {rad[3
              ]:10}')
```

Denne koden leser ut CSV-filen i et litt mer lettleselig format enn det den rå CSV-filen er. Dette gjøres ved å definere en satt minimumsbredde på området ting skal bli skrevet ut på. Dette gjøres i en f-streng med `f'{:10}'` hvor 10 kan være et vilkårlig tall. Dette tallet definerer hvor mange tegn (mellomrom) som skal være bredden på teksten.

```
navn       alder      inntekt    kontor
Marie      30         403279     Stavanger
Fredrik    48         310765     Stavanger
Arne       71         515931     Stavanger
Gunnar     28         436807     Oslo
Martin     41         539709     Oslo
Turid      41         491790     Stavanger
Laila      34         495055     Trondheim
Anna       51         344160     Oslo
Helge      46         451056     Stavanger
Liv        51         676788     Trondheim
Emma       50         647031     Oslo
Heidi      37         607718     Stavanger
Elisabeth  26         401886     Stavanger
```

Men det er ikke alltid det er en liste vi ønsker oss, her er det litt som en liste med lister for hver rad som er lest. Vi kan for eksempel lese det ut som et oppslagsverk, hvor hver rad er et oppslagsverk med posisjonenene øverst i filen som nøkler. Da slipper vi å huske på hvilken kolonne alder eller inntekt var i, vi kan base skrive `rad["inntekt"]` insteden. For å lese inne en CSV-fil på denne måte brukes `csv.`

`DictReader()`. Denne brukes på samme måte som den vanlige `csv.reader()`, men gir oss et oppslagsverk format.

```
with open('.\\ansatte.csv', newline='') as csvfil:
    csv_leser = csv.DictReader(csvfil, delimiter=',')
    for rad in csv_leser:
        print(f'{rad["navn"]:10} {rad["alder"]:10} {rad["
           inntekt"]:10} {rad["kontor"]:10}')
```

```
Marie      30         403279     Stavanger
Fredrik    48         310765     Stavanger
Arne       71         515931     Stavanger
Gunnar     28         436807     Oslo
Martin     41         539709     Oslo
Turid      41         491790     Stavanger
Laila      34         495055     Trondheim
Anna       51         344160     Oslo
Helge      46         451056     Stavanger
Liv        51         676788     Trondheim
Emma       50         647031     Oslo
Heidi      37         607718     Stavanger
Elisabeth  26         401886     Stavanger
```

Det man legger merke til er at vi har mistet den øverste linjen med navnene på kolonnene. For å få med disse igjen må vi legge til en ekstra linje over for-løkken:

```
    print(f'{csv_leser.fieldnames[0]:10} {csv_leser.
       fieldnames[1]:10} {csv_leser.fieldnames[2]:10} {
       csv_leser.fieldnames[3]:10}')
```

```
navn       alder      inntekt    kontor
```

Dette skriver ut `fieldnames`, eller navnene på data feltet eller kolonnene til dokumentet. Mens det kan virke litt kløne p bruke oppslagsverk til dette til å begynne med, så vil verdien bli tydeligere når vi skal skrive til filer.

9.3 Skrive til filer

Akkurat som når vi leser filer, så må vi åpne den først. Men det som er viktig når man skal skrive er at man må gi *tillatelse* til å skrive til filen. Dette er en

beskyttelsesmekanisme for å forhindre at data blir skrevet til en fil man ikke er ment å skrive til. Får å få skriverettigheter må vi legge til 'w', merk at dersom vi ønsker å oppdatere en eksisterende fil og ikke overskrive den bruker vi 'r+'. Hvis man bruker 'w' så vil filen bli opprettet hvis den ikke allerede eksisterer, og hvis den eksisterer så vil den bli overskrevet.

Operatorene for å koble seg til filer er:

r	Les fra fil
w	Skriv til fil. Lager fil hvis den ikke eksisterer. Overskriver
x	Eksklusiv fil laging. Kan kun lage ny fil hvis den *ikke* eksisterer
a	Legg til ekstra innhold på slutten av filen
+	Oppdater innholder i filen

9.3.1 Skrive til tekst filer

Før vi ser mer på CSV-filer så skal vi se litt på tekst filer igjen. Vi kan jobbe med og skrive tekst filer i stor grad på samme måte som vi skriver til terminalen, bare isteden for å bruke `print()` så bruker man `write()`. Hvis vi ser på eksempelet fra forrige kapittel:

```
with open('middagsforslag.txt', mode='r+', encoding='utf-8') as f:
    fil_innhold = f.read()

middager = {}  # tomt oppslagsverk
linjer = fil_innhold.split('\n')
linjer = list(filter(None, linjer))  # fjern eventuelle tomme strenger
for linje in linjer:
    (dag,middag) = linje.split('    ')
    middager[dag] = middag.strip()

print(middager)
```

Dersom vi hadde brukt 'w' så hadde vi endt opp med å overskrive det opprinnelige innholdet i filen. Det vil si at vi ønsker å lese innholdet i filen, og kanskje endre eller oppdatere ting. Vi leser inn dataene på samme måte som vi gjorde i 9.2.2.

```
{'mandag': 'laks og ovnsbakte grønnsaker', 'tirsdag': 'gulrotsuppe', 'onsdag': 'kyllingwok', 'torsdag': '
```

```
   mossaka', 'fredag': 'spansk gazpacho'}
```

Men nå har vi lyst til å endre planen til å ha spaghetti bolognese på mandag. For å endre planen må vi nå skrive til filen. Siden det letteste er å endre innholdet vi ønsker og så skrive over hele filen er det det vi skal gjøre først. Vi har allerede lest inn alle dataene i filen, så nå må vi skrive den oppdaterte listen:

```
[...]
middager["mandag"] = "spaghetti bolognese"

with open('middagsforslag.txt', mode='w', encoding='utf-8'
   ) as f:
    nøkler = list(middager)
    for middag in range(len(middager)):
        f.write(f'{nøkler[i]:<12}{middager[nøkler[i]]}\n')
```

Detførste vi gjør er å åpne filen på nytt. Her ser du at vi bruker 'w' for å åpne filen. Dette er siden vi allerede har hele filinnholdet i minne så kan vi bare overskrive med nytt innhold. Vi `list(middager)` for å få nøklene til oppslagsverket lett tilgjengelig i et løkkbart format, så løkker vi gjennom alle dagene i uken og skriver de inn *linje for linje*. Dette gjøres med `write()`. Det at vi skriver inn linje for linje passer bra i dette tilfellet siden vi løkker gjennom flere ting. Hvis man bare har en liste med linjer (på liste form) så kan man bruke `writelines([liste])`. Det er derimot viktig å merke seg at vi har en `\n` på slutten av stengen vi skriver inn. Dette betyr at her skal det være en ny linje. Datamaskinen har tross alt ikke en enter-tast å trykke på, så vi må fortelle den hvor det skal være en ny linje.

9.3.2 Skrive til CSV-filer

Vi har sett litt på å skrive til tekst filer, men det er ofte litt mer nyttig å skrive til en CSV-fil. Disse filene er også litt lettere å håndtere siden det er et mer "fast format", så lenge vi ignorer alle variantene. Vi bygger igjen på eksempelet fra forrige kapittel, 9.2.3:

```
with open('.\\ansatte.csv', newline='') as csvfile:
    csv_leser = csv.DictReader(csvfile, delimiter=',')
    print(f'{csv_leser.fieldnames[0]:10} {csv_leser.
       fieldnames[1]:10} {csv_leser.fieldnames[2]:10} {
       csv_leser.fieldnames[3]:10}')
    for rad in csv_leser:
```

```
5           print(f'{rad["navn"]:10} {rad["alder"]:10} {rad["
                inntekt"]:10} {rad["kontor"]:10}')
```

La oss si at bedriften har hatt en kjempe på så alle får en lønnsøkning på 10%. Vi må nå oppdatere CSV-filen vår med de nye lønnstallene. Vi har allerede lest inn dataene, så nå må vi bare oppdatere dem. Først lager vi en funksjon for å oppdatere dataene til de ansatte:

```
1 def lonn_okning(ansatte, okning):
2     oppdaterte_ansatte = list()
3     for ansatt in ansatte_dict_list:
4         ny_lonn = int(round(int(ansatt['inntekt'])* (1 +
              okning), 0))
5         ansatt['inntekt'] = str(ny_lonn)
6         oppdaterte_ansatte.append(ansatt.copy())
7     return oppdaterte_ansatte
```

Denne funksjonen tar inn to argumenter, en liste med oppslagsverk og en lønnsøkning som desimaltall. Men for å kunne bruke denne funksjonen så må vi den noen data. Vi forenkler da koden vi brukte for å åpne den tidligere, siden vi ikke har behov for å skrive ut dataene:

```
1 with open('.\\ansatte.csv', mode='r',newline='') as
      csvfile:
2     csv_dictleser   = csv.DictReader(csvfile, delimiter=',
          ')
3     ny_liste = lonn_okning(csv_dictleser, 0.1)
```

Her åpner vi da filen i "read" form, siden vi ikke skal skrive til den riktig enda. Først tar vi bare å leser ut innholdet, før vi sender det til funksjonen vi akkurat lagde. Men, vi må jo skrive noe tilbake til filen, ellers får vi jo ikke lagret lønnsøkningen:

```
1 felt = ['navn', 'alder', 'inntekt', 'kontor']
2 with open('.\\ansatte.csv', mode='w',newline='') as
      csvfile:
3     csv_dictskriver = csv.DictWriter(csvfile, delimiter=',
          ', fieldnames=felt)
4     csv_dictskriver.writeheader()
5     csv_dictskriver.writerows(ny_liste)
```

Her har vi da enda en åpning av den samme filen etter at den forrige ble lukket.

Grunne for dette er at vi risikerer å bare delvis overskrive eller på andre måter få kluss i dataene hvis vi prøver å skrive inn i en allerede eksisterende fil. Derfor åpner vi den nå i "w" modus, siden dette sletter alt innhold i filen. Vi har også definert `felt`-listen. Denne blir gitt til "skriverhodet" som tilhører CSV-filen, slik at den vet hva de forskjellige kolonnene skal hete. Dette blir så skrevet til filen før vi skriver inn den oppdaterte listen med ansatte. På samme måte så kan man fjerne eller leggetil ansatte. Men hvis man *kun* skal legge til, så kan man bruke *append* insteden for write. Dette vil la deg skrive inn data helt i slutten av filen, hvilket er perfekt hvis man bare skal legge til noe nytt.

9.4 Avansert eksempel: Tekst fra fil

Nå som vi har sett på det å skulle lese fra og skrive til filer, la oss se på et litt mer samensatt eksempel. Vi skal bygge videre på tekstbehandlingsfunksjonene vi lagde i forrige kapittel 8.6. Videre skal vi benytte filbehandling for å lese inn tekst data for behandling, for så å lagre resultatet i en CSV-fil.

Det første vi skal gjøre er å lese inn innholdet fra en større tekst fil. I dette tilfellet så skal vi lese inn Alice in Wonderland. Vi starter da med en standard innlesing-funksjon:

```
with open('Alice.txt', mode='r', encoding='utf-8') as f:
    fil_innhold = f.read()
```

For å sjekke at vi har lest inn riktig innhold så kan vi printe ut lengden på teksten

```
print(f'Lengden til filinnholdet: {len(fil_innhold)}')
```

```
Lengden til filinnholdet: 163817
```

Nå som vi vet at innholdet virker korrekt, så kan vi starte med å behandle det. Vi bruker funksjonene fra 8.6 til å lage noen nye funksjoner. Vi bruker her to innebygde biblioteker, `string` og `Collections`. `string` lar oss bruke `string.punctuations` som er en streng med alle tegn brukt for å starte eller avslutte tekster. `Collections` bruker vi for `Collections.Counter`, hvilket gir oss en enkel måte å telle ting på, siden det teller opp hver gang vi legger til et element som allerede er i variabelen.

```
import string
from collections import Counter

```

```
def tekst_kompleksitet(tekst):
    return setnings_lengde_snitt(setning_deler(tekst))

def ord_antall(input_dict):
    total = 0
    for key in input_dict.keys():
        total += int(input_dict[key])
    return total

def finn_ord_frekvens(file_path):
    with open(file_path, mode='r', encoding="utf-8") as f:
        file_text = f.read().lower()
        wordcount = Counter(file_text.translate(str.
           maketrans('','',string.punctuation)).split())
    return wordcount

def finn_vanlige_ord(dicts_list, limit):
    entry_list = []
    temp_list = []
    dictionaries_counter = 0
    for counter in dicts_list:
        dictionaries_counter += 1
        temp_list = sorted(counter, key=counter.get,
           reverse=True)
        del temp_list[limit:]
        entry_list += temp_list.copy()
    counterd = Counter(entry_list)
    common = counterd.most_common(limit)
    common_words = list()
    for i in common:
        common_words.append(i[0])
    return common_words
```

Vi har fire helt nye funksjoner her, `tekst_kompleksitet()`, `ord_antall()` `finn_ord_frekvens()` og `finn_vanlige_ord()`. `tekst_kompleksitet()` er bare en mer kopakt versjon av hvordan vi fant den gjennomsnittlige setningslengden i forrige kapittel. Dette gjør det bare litt enklere å bruke, samtidig som det gjør det tyeligere hva formålet er. `ord_antall()` er en liten funksjon som teller opp hvor mange ord det er totalt i en tekst. Den tar inn et oppslagsverk og teller over hvor mange tall som er registrert totalt. `finn_ord_frekvens()` er en raffinering av `ord_teller()` fra forrige

kapittel som lar oss lettere telle ordene i en lengre tekst, samtidig som den er bedre på å fjerne tegn i teksten. `finn_vanlige_ord()` er en videreutvikling av ordskyen som ble nevnt i forrige kapittel. Her tar det da inn en liste med oppslagsverk for så å finne de 10 vanligste ordene blant alle de som oppslagsverkene som ble sendt inn. Grunnen til å bruke en liste i steden for å bare ta inn et oppslagsverk er for å kunne behandle mange filer på en gang, hvilken vi vil komme tilbake til litt senere i eksempelet.

Til slutt så kan vi bruke alt dette for å få en liten analyse av teksten:

```
aiw_ordfrekvens = finn_ord_frekvens('Alice.txt')

print(f'Lengden til filinnholdet Alice in Wonderland: {len(fil_innhold)}')
print(f'Gjennomsnittlig setningslengde i Alice in Wonderland: {tekst_kompleksitet(fil_innhold):.2f}')
print(f'Totalt antall ord i Alice in Wonderland: {ord_antall(aiw_ordfrekvens)}')
print(f'Totalt antall unike ord i Alice in Wonderland: {len(aiw_ordfrekvens)}')
print(f'De 10 vanligste ordene i Alice in Wonderland: {finn_vanlige_ord([aiw_ordfrekvens], 10)}')
```

```
Lengden til filinnholdet Alice in Wonderland: 163817
Gjennomsnittlig setningslengde i Alice in Wonderland: 16.58
Totalt antall ord i Alice in Wonderland: 29390
Totalt antall unike ord i Alice in Wonderland: 3949
De 10 vanligste ordene i Alice in Wonderland: ['the', 'and', 'to', 'a', 'of', 'she', 'it', 'said', 'in', 'alice']
```

Det er jo vel og bra å ha dette gjort, men hva hvis vi ønsker å finne dette på nytt igjen? Det er jo litt sløsete å kjøre analysen hver gang vi skal hente ut data, så la oss lagre det. For dette så kan vi bruke en CSV-fil, men før vi gjør det så må vi strukturere dataene våre for å bli skrevet inn. Det første vi må gjøre er å ha analysen vår som variabler.

```
aiw_lengde = len(fil_innhold)
aiw_kompleks = tekst_kompleksitet(fil_innhold)
aiw_ord = ord_antall(aiw_ordfrekvens)
aiw_unike_ord = len(aiw_ordfrekvens)
aiw_vanlige_ord = finn_vanlige_ord([aiw_ordfrekvens], 10)
```

```
6 aiw_vanlige_ord_streng = str(aiw_vanlige_ord).translate(
    str.maketrans('','',"[,]'"))
```

Det er verdt å merke at jeg bruker `string.translate` og `str.maketrans` for å fjerne alle de uønskede tegnene i `aiw_vanlige_ord_streng`. Grunnen for å gjerne tegnene, spesielt komma, er at det vil skape problemer i CSV-filen siden komma er separatoren for kolonner.

Det neste vi må gjøre er å definere hva navnene på kolonnene våre skal være:

```
1 label = ['fil', 'tegn', 'ord', 'setningslengde', '
    unike_ord', '10_vanligste_ord']
```

Og siden `csv.DictWriter()` er veldig grei å bruke for å skrive inn data, så konverterer vi det til et oppslagsverk og skriver til fil:

```
1 csv_dicts = {label[0]:'Alice.txt', label[1]:aiw_lengde,
    label[2]:aiw_ord, label[3]:f'{aiw_kompleks:.4f}', label
    [4]:aiw_unike_ord, label[5]:aiw_vanlige_ord_streng}
2
3 with open('filbehandling.cvs', mode='w', encoding='utf-8')
     as f:
4     skriver = csv.DictWriter(f, label)
5     skriver.writeheader()
6     skriver.writerow(csv_dicts)
```

Legg merke til at med unntak av `'Alice.txt'` så er alle variabler. Det vil gjøre det lett å bruke koden på nytt senere, hvis man skal jobbe med et lignende prosjekt.

10 Tillegg A - 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 Python-koden og oversette dette til maskininstrukser som datamaskinen så kan gjennomføre. Du må altså kunne både *skrive* og *kjøre* Python-kode.

Ettersom at Python utvikles som et fritt programvareprosjekt, kan hvem som helst lage sine egne programmer for å jobbe med 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 Python-kode og resultatene de gir. Disse eksemplene derimot, vil vises i en generell form og vil ikke være knyttet opp mot spesifikk programvare.

10.1 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 er i 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. I tillegg blir Python 2 faset ut i slutten av 2020.

10.2 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 programpakka *Anaconda Distribution*[2]. 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 4. 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 lukke dette vinduet fullstendig, ved å klikke et par ganger på korset øverst til høyre i hjelpevinduet.

10.2.1 Python på datamaskin: IDLE

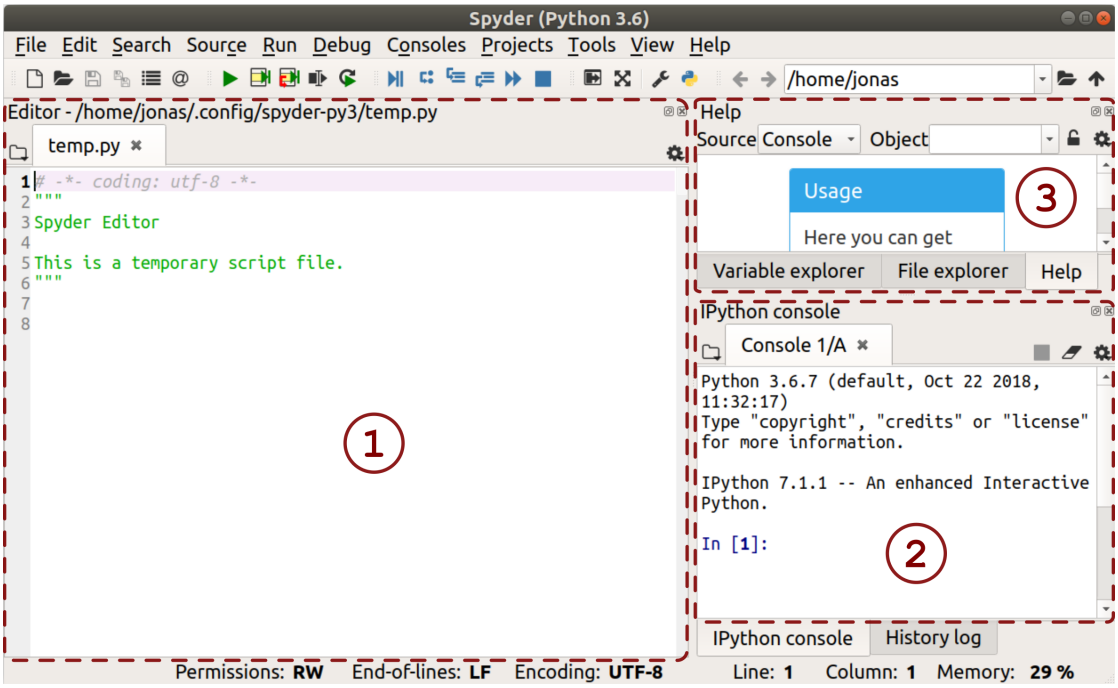
Et alternativ til å laste ned Anaconda, er å laste ned Python direkte fra utviklersiden `Python.org`[3]. Her får du en helt grunnleggende Python-installasjon, tilleggspakker for å for eksempel drive med turtleprogrammering må isåfall installeres separat. I tillegg til Python-installasjonen får du redigeringsprogrammet IDLE. Sammenlignet med Spyder er IDLE langt mer minimalistisk, men dette kan fungere godt for nybegynnere.

10.3 Python i nettleser: Trinket

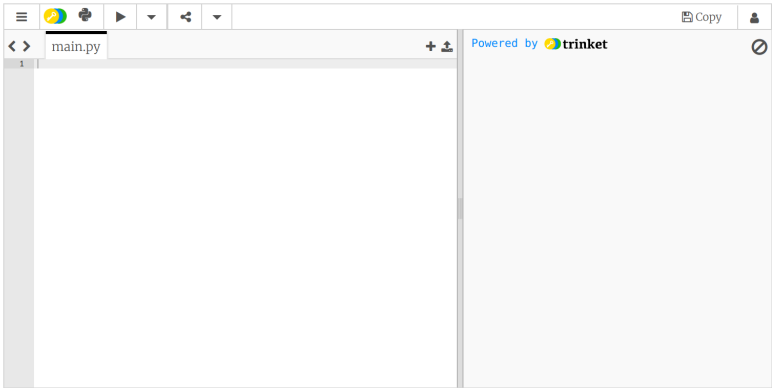
Et alternativ til å installere programvare på egen maskin er å programmere i nettleseren sin. Det finnes flere ulike nettsider som lar deg både skrive og kjøre

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

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



Figur 4: En skjermfangst av hvordan Spyder kan se ut ved første bruk. I tillegg til menyer og en verktøylinje på toppen, består programmet av ulike vinuder. Redigeringsvinduet (1) brukes til å skrive selve Python-koden vår, konsollen (2) er der vi får resultatene når vi kjører et program, og (3) er et hjelpevindu hvor vi kan få tilleggsinformasjon.



Figur 5: En skjermfangst av nettsiden `trinket.io/python3`. Vinduet til venstre er der vi kan skrive koden. Vinduet til høyre viser resultatene av en kjøring.

Python-kode 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

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

Kommer du rett inn i en interaktiv sesjon hvor du kan skrive og kjøre Python-kode. En skjermfangst vises i Figur 5.

Trinket kan brukes gratis, uten å lage bruker. Imidlertid 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 flere fordeler. Det kan være veldig praktisk å lage en gratisbruker for lærer. I praksis kan det være utfordringer med at elever lager brukere grunnet personvernhensyn.

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

10.3.1 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 Goog-

le. Coding with Chrome fungere med både Python og Javascript, men også det blokkbaserte språket blockly.

10.4 Python på nettbrett

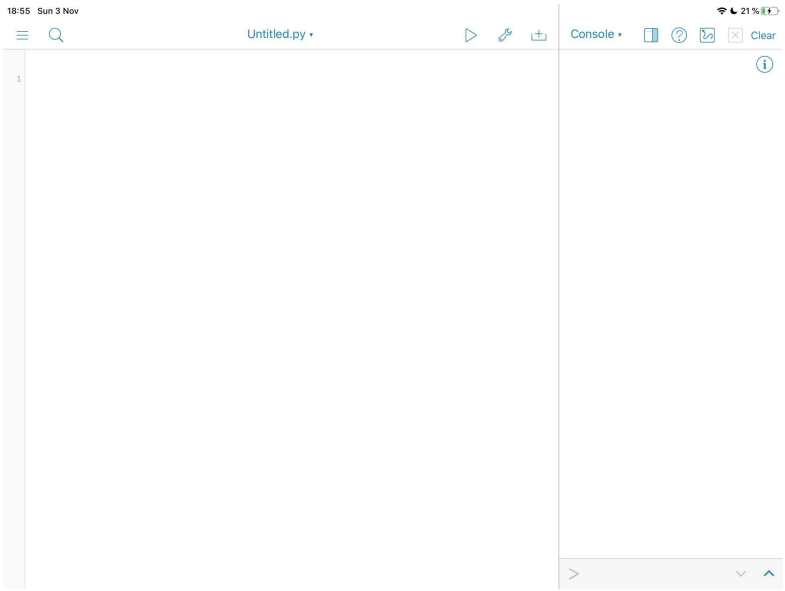
For å programmere på nettbrett er nettbaserte løsninger fortsatt en god mulighet. Det finnes også visse applikasjoner som lar det skrive og kjøre Python-kode på nettbrett eller telefon, dessverre er ikke alle disse av veldig god kvalitet, eller er like nybegynnervennlige, men noen av dem er veldig gode.

10.4.1 Python på iPad

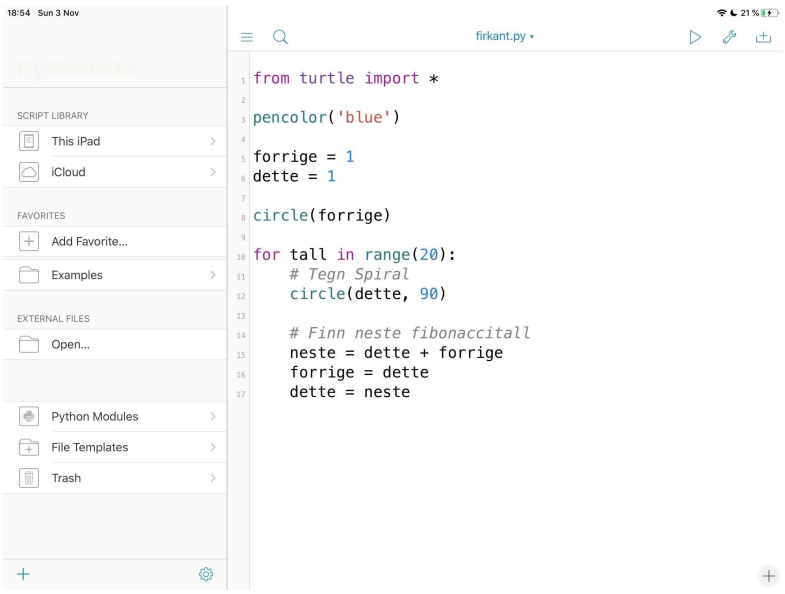
Om du er på iPad kan vi anbefale applikasjonen Pythonista 3, som er godt utviklet, og har et intuitivt brukergrensesnitt. I motsetning til de andre løsningene vi nevner har Pythonista kun en lisensiert versjon som koster penger. Pythonista er intuitiv å bruke, har god syntaks-utheving, og er kompatibel med alle de vanlige eksterne Python-bibliotekene.

Pythonista gjør det enkelt å dele filer med andre enheter, og er kompatibel med deling via bl.a. Showbie, dersom man bruker det.

I tillegg kommer Pythonista med mange ferdige eksempler på applikasjoner og spill, med åpen kildekode som man kan leke seg med som en bonus.



Figur 6: Et tomt Pythonista-vindu - koden skrives inn i området til venstre, og kjøres med play-knappen. Outputen kommer i vinduet til høyre, men man kan også ha innstillinger slik at hvert av vinduene fyller hele skjermen, og man bytter mellom dem.



Figur 7: De tre strekene i høyre hjørne gir tilgang til en meny der man kan navigere mellom mapper, og justere innstillinger som farge og fontstørrelser.

11 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.

Algoritme Dette er en oppskrift 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. De er i tillegg også spesielt viktige innen programmering, fordi datamaskiner egentlig er ganske dumme og derfor kun kan 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*.

Array En sekvens med tall som vi, i motsetning til *lister*, kan regne med. Det er imidlertid mulig å konvertere en *liste* med tall til en array. Å regne med et helt *array* av gangen kalles en vektorisert beregning.

Betingelse En betingelse gjør det mulig for datamaskinen å gjøre forskjellige ting basert på konteksten. Betingelser er basert på *boolsk logikk*, og helt nøyaktige – enten er noe sant, eller så er det ikke sant. På engelsk kalles dette gjerne for *if statements*, som kan oversettes med *hvis-utsagn*. Man kan også kalle dem for *betingelsessetninger*. Betingelser 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.

Boolsk logikk Datamaskiner er basert på om en tilstand enten er sann eller usann. Man kan lage egne tilstander ved å kombinere tilstander som enten er

sanne eller usanne ved å bruke de logiske operatorene *og*, *eller* og *ikke*.

Funksjon 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 våre 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.

Kodelinje 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*.

Kommentar En kommentar kommer bak et #. Ved å legge til en kommentar i programmet kan man lettere holde oversikt over hva kodelinjene gjør og betyr. En kommentar vil ikke ha noe å si for koden i programmet, og vil ikke vises i output-en.

Liste En *variabel* som i stedet for å inneholde bare et element, kan innholde flere elementer. Listeelementer er sortert i en bestemt rekkefølge, men listen kan modifiseres og endres. En *liste* kan inneholde både tekststrenger og tallverdier.

Løkke 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*.

Program 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.

Variabel 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.

Register