Programmering i realfagene

Et kræsjkurs i Python for lærere i videregående opplæring

kodeskolen simula

kodeskolen@simula.no

Dette kompendiet er kursmateriale for de to første kursdagene i kurs om programmering for realfagslærere. Målet med kompendiet er å gi en innføring i programmering med Python som er tilstrekkelig til å begynne å se på problemstillinger som er interessante og nyttige for realfagsundervisning.

Kompendiet antar at leseren har liten, eller ingen, tidligere er faring med programmering, og prøver derfor å gå grundig igjennom konseptene. Om du fremdeles syns kompendiet går for fort frem, si gjerne ifra, så skal vi finne alternative kilder til dere. På den andre siden av mynten kan du skumme raskere igjennom kompendiet om du allerede kan programmere. Isåfall anbefaler vi spesielt å se på eksempler og oppgaver.

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

Innhold

1 Vi setter igang

1.1 Nødvendig Programvare

Å programmere går ut på å instruere datamaskinen til å utføre de stegene og oppgavene vi ønsker at den skal gjøre. Disse instruksene gir vi ved å skrive *kode*, som rett og slett er instrukser gitt på en form som datamaskinen forstår og kan utføre.

Når vi skriver kode må vi velge oss et bestemt *programmeringsspråk*. Akkurat som et vanlig språk finnes det hundrevis av forskjellige å velge blant, og til og med forskjellige dialekter og varianter.

I dette kurset fokuserer vi på språket Python. For å programmere i Python trenger vi å installere to ting: selve programvaren Python, og et program til å skrive koden i. Den enkleste måten å installere dette på egen maskin er å installere *Anaconda Distribution*, da får vi selve Python, ekstra pakker som kan være nyttige i realfag, samt et koderedigeringsprogram.

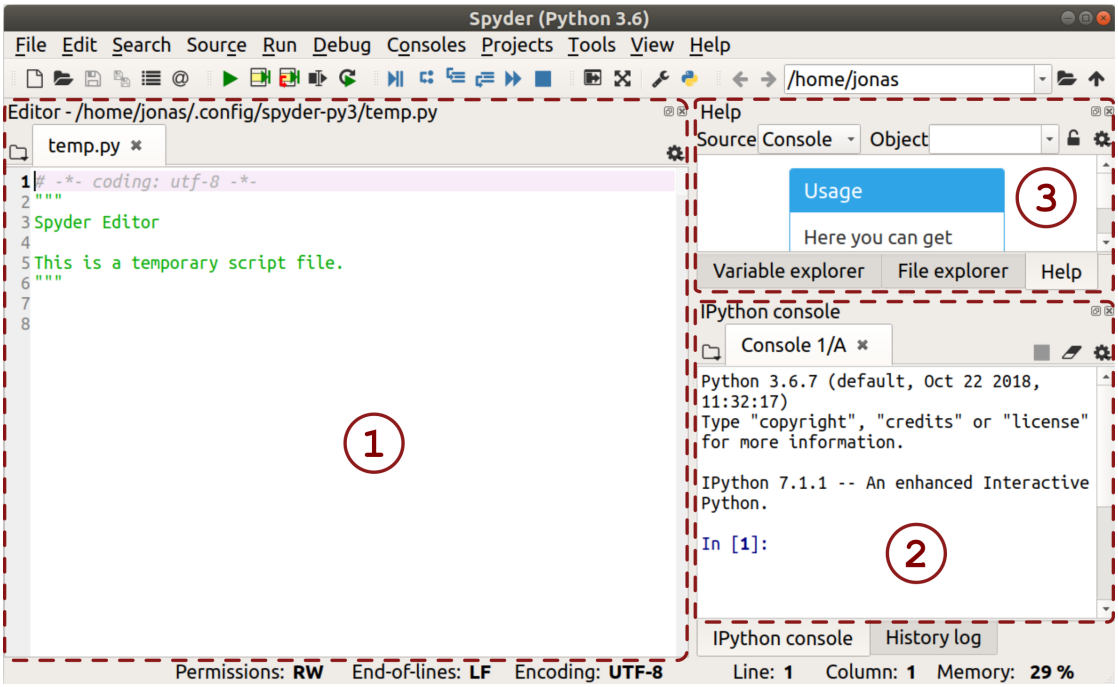
For å installere kan du gå inn på

- `https://www.anaconda.com/distribution/`

Velg versjon 3.7 av Python, og pass på å velg riktig variant for ditt operativsystem (Windows/Mac/Linux).

Når du har installert *Anaconda* vil du ha fått med et program som heter *Spyder*, og det er dette programmet vi kommer til å bruke i kurset til å skrive, og kjøre, Pythonkode. Spyder er bare ett av mange mulige programmer man kan bruke til å jobbe med Python. Vi anbefaler det hovedsakelig fordi det installeres automatisk gjennom Anaconda, og at det derfor gjerne er det enkleste alternativet å komme igang med.

Første gang du åpner Spyder ser det hele gjerne litt komplisert ut, du kan se en skjermfangst i Figur 1. 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 større vinduet på venstre side av skjermen (1) er der vi skriver koden vår, det kalles gjerne for en *Editor* på engelsk. Det mindre vinduet nede til venstre (2) kalles for en *Console*, og det er her resultatene fra



Figur 1: En skjermfangst av hvordan Spyder ser ut ved første bruk.

programmene våres vil vises. Vinduet øverst til høyre (3) er et hjelpevindu som kan gi tilleggsinformasjon.

Før vi setter igang med selve programmeringen vil vi anbefale deg å gjøre et par små endringer til Spyder:

- Hjelpevinduet er gjerne mer distraherende enn det er til hjelp for nybegynnere. Vi anbefaler derfor rett og slett å fjerne det. Dette gjøres ved å klikke på det lille krysset øverst til høyre av hjelpevinduet. Da sitter vi kun igjen med editor-vinduet til venstre, og console-vinduet til høyre.
- Under innstillinger (*Tools* → *Preferences* → *Run*) huk av for *"Remove all variables before execution."* Dette er ikke strengt nødvendig, men kan gjøre det litt lettere å finne feil i koden vår senere.

1.2 Ditt første program

Om installeringen av Anaconda ser ut til å ha vært vellykket er du nå klar for å skrive og kjøre Pythonkode og det er på tide å lage ditt første program.

For å lage et program skriver du koden din inn i editor-vinduet til venstre. Før vi kjører koden må vi lagre programmet vårt. Dette fungerer som lagring av filer i de fleste programmer, velg *"Save as"* (ev bruk Ctrl+Shift+S). Alle python programmer lagres som filer med ".py" endelse. Her kan du for eksempel velge navnet `hello.py`.

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.

I Python er dette programmet ganske enkelt, det er faktisk kun én enkelt linje med kode:

```
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. Dette gjøres i Spyder ved å klikke på "Run"-knappen du finner i verktøylinja i toppen, eller ved å klikke på F5.

Om du har skrevet koden riktig, og alt er fungerer som det skal, så skal du nå få ut beskjeden i console-vinduet til høyre. Om noe er galt 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. Spør om hjelp om det fortsatt ikke fungerer.

1.3 Print

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

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

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

Merk at i Spyder kan fargene være litt ulike fra de vi viser i eksemplene i dette kompendiet. Akkurat hvilke farger de forskjellige delene av koden blir er ikke så viktig. Fargene er der bare for å lettere skille mellom de ulike delene av koden. Du kan og endre på fargeinnstillingene i Spyder om du foretrekker andre fargevalg.

1.4 Et eksempelprogram

"Hello, World"-programmet du nå har skrevet er på mange måter det aller enkleste programmet vi kan lage. Det er mest en test for å sjekke at alt er satt opp riktig på datamaskinen. Nå går vi igjennom et litt større eksempel for å vise hvordan et kort program gjerne kan se ut. Målet her er bare å få en litt bedre forståelse av et program, og vi kommer til å dekke alle detaljene i koden etterhvert utover i dette kompendiet.

```python
# Importer nødvendig funksjonalitet
from pylab import pi

# Sett radius
radius = 11

# Regn ut verdier
omkrets = 2*pi*radius
areal = 4*pi*radius**2
volum = 4*pi*radius**3/3

# Skriv ut resultater
print(f"En fotball med radius på {radius} cm")
print(f"Har en omkrets på {omkrets:.1f} cm")
print(f"Et overflateareal på {areal:.1f} cm²")
print(f"Og et volum på {volum/1000:.1f} liter")
```

Når vi kjører dette programmet så vil datamaskinen gjøre et par beregninger, og vi får skrevet noen linjer med informasjon ut. I dette kompendiet kommer vi ofte til å vise resultatet av en kjøring i et eget vindu litt under eksempelkoden, som følger:

```
En fotball med radius på 11 cm
Har en omkrets på 69.1 cm
Et overflateareal på 1520.5 cm²
Og et volum på 5.6 liter
```

Det vi her kaller et "program" er et sett med kodelinjer. Når vi kjører koden vil disse tolkes linje for linje, fra toppen og nedover. Hver linje svarer til en bestemt instruks som datamaskinen vil tolke og gjennomføre.

Merk at et par av linjene vi har skrevet starter med et `#`-symbol og er markert i gult. Slike linjer kaller vi *kommentarer*. Disse linjene er egentlig ikke kode, og de påvirker ikke oppførselen til programmet vårt, istedet er de der for å gjøre det lettere for oss å holde oversikt over hva de ulike delene av koden gjør. Slike kommentarer er ofte nyttige å skrive. Spesielt om man skal dele koden sin med andre.

Den første kodelinjen som faktisk gjør noe er `from pylab import pi`. Denne linjen har vi med fordi vi trenger å bruke tallet π, som Python ikke kjenner til fra før.

Derimot kan vi *importere* dette tallet fra en ekstrapakke.

På den neste kodelinjen setter vi radiusen til en kule til å være 11, med linjen `radius =11`. Her *definerer* vi en variabel. Vi velger 11 fordi en vanlig fotball er omtrent 11 cm. Vi kan ikke inkludere enheten (cm) her, så det må vi bare huske på, og tolke svarene våres som kvadrat- og kubikcm til slutt.

På de neste tre kodelinjene regner vi oss frem til omkretsen, overflatearealet og volumet til fotballen. Dette gjør vi ved å definere nye variabler, men istedenfor å sette dem til en verdi vi skriver ut selv, så ber vi Python regne ut disse for oss.

Til slutt har vi fire kodelinjer som bruker `print`-funksjonen, som vi allerede har sett et eksempel på. I dette tilfellet er bruken av `print` noe mer komplisert, men vi forklarer dette senere. Hovedpoenget er at vi bruker `print` til å skrive ut resultatet av beregningene på en måte som er oversiktlig.

Programmet som er vist frem her gjør altså noen enkle geometriske beregninger og viser frem resultatet. Når vi først har skrevet koden kan vi kjøre den så mange ganger vi måtte ønske. Vi kan nå for eksempel gå og endre radiusen til 12, 15, eller 20, og kjøre på nytt med ett enkelt tastetrykk.

1.5 Kunsten å lese og skrive kode

Når man er helt fersk til programmering kan det være vanskelig å lese, skrive, og forstå kode. Ikke bare er det et helt nytt språk, men det er gjerne en helt ny måte å tenke på i tilegg. Det betyr at det fort kan bli for mye og at det føles uoversiktlig og uoverkommelig.

Enten man leser eller skriver kode er det viktig å jobbe seg frem stegvis, linje for linje. Om man setter seg litt fast er det ofte nyttig å stille seg selv spørsmål, eller å diskutere med en annen. Dette er spesielt også viktig når elever setter seg fast.

Her er noen eksempler på spørsmål som kan være nyttige for å sette igang tankene og diskusjonen:

- Hva er baktanken til dette programmet?
- Hva prøver det å oppnå? Hva er de grove stegene for oppnå dette målet?
- Hvorfor gjøres ting i denne rekkefølgen?

- Hva *prøver* vi å oppnå med akkurat denne kodelinjen?
- Hva er det denne kodelinjen gjør?

1.6 Om det å gjøre feil

Etterhvert som du jobber deg igjennom dette kompendiet og prøver deg på eksempler og oppgaver kommer du til å gjøre mye feil. Sånn er det bare rett og slett. Det å gjøre feil er en utrolig stor del av å programmere, og sånn er det ikke bare for nybegynnere. Erfarne programmere gjør også utrolig mye feil, de bare lærer seg å bli mer effektive på å finne ut av feilene de har gjort og løse opp i dem raskere. Det viktigste budskapet vi kan gi er: Ikke vær redd for å gjøre feil.

Når du programmerer er det to hovedkategorier med feil du kan gjøre. Du kan lage kode som ikke kjører i det heletatt eller som kræsjer halvveis igjennom, og du kan lage kode som kjører fint, men som gir feil resultater.

Datamaskinen er egentlig ganske dum, og om den ikke skjønner nøyaktig hva du ønsker at den skal gjøre, så vil den rett og slett nekte å gjøre noe som helst. Resultatet er et program som ikke kjører i det heletatt eller kræsjer halvveis igjennom. Når dette skjer vil Python prøve å gi deg informasjon om hva som er galt i form av en feilmelding. Disse feilmeldingene er kryptiske og skumle for nybegynnere, men prøv gjerne å lese dem når ting går galt. De kan kanskje gi deg et hint om hvor du bør se, og etterhvert som du gjør den samme feilen flere ganger lærer du å kjenne igjen feilmeldingen.

Den andre typen problem høres kanskje bedre ut, at koden kjører, men gir feil resultater. Men disse feilene kan være langt farligere, fordi det da kan være at vi *tror* programmet fungerer som det skal, og vi stoler på feil eller dårlige resultater. Det er for eksempel mye forskning som har blitt tilbakekalt grunnet kode som har vist seg å være feil. Et mer dramatisk eksempel er kollapsen av Sleipner A plattformen i 1991. Denne ble bygget underdimensjonert i understellet grunnet dårlige resultater fra feil kode. Det er derfor viktig å ikke stole blindt på det som spyttes ut av programmene vi lager, men å stole på intuisjonen, etterprøve svarene våres og se at ting er internt konsekvent—her kan det trekkes mange paralleler til matematiske beregninger og utledninger. Dette er et introduksjonskurs, så vi skal ikke legge så altfor mye vekt på hvordan man kan skrive pålitelig og feilfri kode. Men vi skal prøve å ha et gjennomgående fokus på å sjekke om resultatene våres er rimelige.

2 Variabler og Utregninger

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.

2.1 Definere variabler

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

```
1 radius = 11
2 masse = 0.5
3 avstand = 2500
```

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 om å huske på en radius, en vekt, og en avstand. Informasjonen lagres i minnet på datamaskinen, og vi får et navn i programmet vi kan bruke til å hente informasjonen ut og bruke senere.

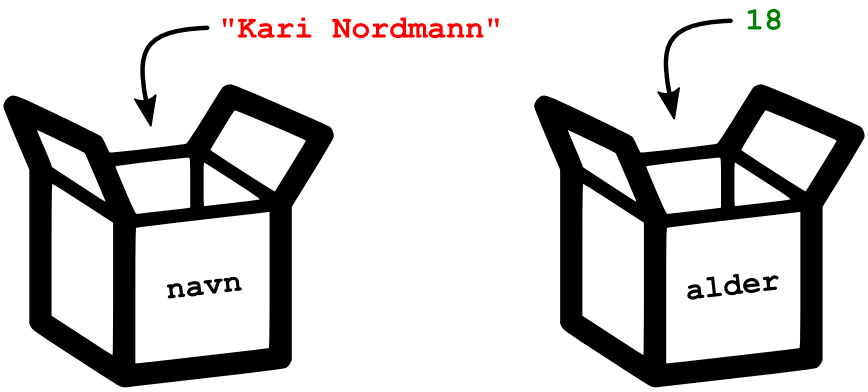
Når vi oppretter en variabel får den et *navn* og en *verdi*. Navnet er det vi skriver på venstre side av likhetstegnet (`radius`/`masse`/`avstand`), og verdien er det vi skriver på høyre side.

For å forstår variabler kan man tenkte på dem som en esker som brukes til oppbevaring. Verdien til variabelen er det samme som innholdet, mens navnet skrives utenpå. Datamaskinen tar seg av ansvaret med å arkivere boksen for oss i et stort lagerhus (datamaskinens minne), men kan enkelt finne dem frem igjen når vi ønsker den, takket være navnet.

I tilegg til et navn og en verdi har enhver variabel også en *type*. Dette er fordi variabler kan være langt mer enn bare tall. Vi kan for eksempel opprette variabler som inneholder tekststrenger:

```
1 navn = "Kari Nordmann"
2 alder = 18
```

Her oppretter vi to forskjellige variabler, den første inneholder en tekststreng, mens den andre innholer et heltall.



Figur 2: Variabler kan beskrives som esker der vi oppbevarer, eller lagrer, informasjon. Alle variabler har et *navn*, som vi bruker til å referere til variabelen. De har en *verdi*, som er selve innholdet i variabelen. I tilegg kan vi snakke om variabelens *type*, som avhenger av hva slags innhold variabelen er, f.eks tekst eller tall.

Merk at det er du som programmerer, som velger navnet på variablene dine helt fritt. De eneste reglene som gjelder er at man ikke kan bruke navn som er de samme som innebygde funksjoner i Python (f.eks kan du ikke lage en variabel som heter `print`), og navnet kan ikke inneholde mellomrom, eller spesielle tegn som parenteser og lignende.

Mange programmeringsspråk klager på bruke av æ, ø og å, og et vanlig tips er derfor alltid å programmere på engelsk. I Python 3 fungerer derimot disse knirkefritt, og det er fritt frem å programmere på norsk om man ønsker det.

2.2 Bruke variabler

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

```
1 person = "Ola"
2
3 print('Hei på deg')
4 print(person)
```

```
Hei på deg
Ola
```

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, og det er dette innholdet som skrives ut.

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

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

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

Se spesielt nøye på koden her. Hvilke deler av utskriften er tekst, og hvilken er variabler? Her kan fargene være til hjelp.

Variabler er viktige nesten uansett hva slags program vi skal lage, og det er derfor viktig å beherske begrepene vi bruker rundt variabler. Her er det altså lurt å være å være konsekvent i bruken av at man *definerer* eller *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.,

2.2.1 Litt mer om variabeltyper

Som vi har nevnt har alle variabler en bestemt *type*. Når vi oppretter en variabel vil Python selv velge hva slags type objekt det er vi lager. Dette er ulikt fra noen andre programmeringsspråk, hvor man eksplisitt må spesifisere typen. Fordelen med at Python gjør dette automatisk er at det blir mindre å skrive, og det kan være lettere for nybegynnere. Ulempen er at det kan bli forvirrende om man ender opp i en situasjon der typene er viktige. Vi erfarer at å bruke for mye tid på å snakke om typer for de som er nye til programmering kan føre til mer forvillelse

enn oppklaring, og vi kommer derfor ikke til å snakke for mye om typer unntatt der det er viktig.

Uansett kan det være nyttig å vite at man kan sjekke typen av et objekt med funksjonen `type()`. Slik som dette:

```
navn = "Ola"
alder = 18
høyde = 172.3

print(type(navn))
print(type(alder))
print(type(høyde))
```

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

Her ser vi at variabelen `navn` har type `str`. Dette er kort for "string". I informatikken kaller vi gjerne tekst for tekststrenger, og `str` variabler er altså tekststrenger. Vi ser at variabelen `alder` ble `int`. Dette er kort for "integer" som betyr heltall på engelsk. Til slutt har vi `høyde` som blir `float`. Dette kommer av at desimaltall på datamaskinen kalles for *flyttall*.

Det finnes langt flere typer variabler enn disse tre i Python, men vi ønsker ikke å bruke for mye tid på dette nå, så vi går ikke videre inn på dette.

2.3 Utregninger

Vi vil nå begynne å se på hvordan vi kan regne med variabler i Python. En av de store fordelene med datamaskiner er at de er så raske til å regne, de er rett og slett *tallknusere*. Man kan derfor godt bruke Python som kalkulator.

La oss gå tilbake og se på eksempelet vi brukte i starten av dette kompendiet. Vi ønsker å regne ut hvor mange liter 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 følgende utregning:

```
pi = 3.14
radius = 1.1  # dm
volum = 4*pi*radius**3/3  # liter
```

Her har vi skrevet tre kodelinjer som gjennomfører utregningen vi ønsker. De to første linjene definerer to variabler: `pi` og `radius`, som henholdsvis skal ha verdiene 3.14 og 1.1. Vi definerer π ettersom at Python ikke har denne verdien innebygd. Merk at vi vi bak radius skriver en *kommentar*, med `# dm`. Husk at slike kommentarer ikke påvirker selve koden, det er der for å hjelpe oss å huske på hvilke enheter vi bruker.

Når vi har definert de to variablene våres, regner vi ut volumet fotballen ved å skrive formelen `volum =4*pi*radius**3/3`. La oss forklare denne koden litt mer i detalj. I selve formelen bruker vi `*` (asterisk/stjerne) for å gange og `**` for *opphøyd i*. Altså svarer selve utregningen til den matematiske formelen

$$\texttt{volum} = \frac{4}{3}\pi r^3.$$

Siden vi begynner kodelinjen `volum =` så definerer vi faktisk en ny variabel her, med navn `volum`. Innholdet i variabelen er resultatet av utregningen.

Hvis du skriver inn denne koden på din egen maskin, og kjører den, så vil du oppdage at det skjer ingenting. Eller kanskje mer riktig: vi merker ikke at det skjer noe. Dette er fordi vi ikke har brukt `print` i koden vår. Her gjennomfører altså datamaskinen utregningen vår, men siden vi ikke spør om å faktisk få se resultatet, så viser den det ikke frem.

Derfor må vi be datamaskinen også skrive ut svaret:

```
print(volum)
```

```
5.5724533333333355
```

Dette er verdien Python har regnet ut for oss. og svaret vårt, altså inneholder fotballen 5.6 liter luft. Akkurat nå er ikke utskriften veldig pen, den inneholder blant annet fryktelig mange desimaler. Vi kommer tilbake til hvordan man kan ha bedre kontroll på utskriften.

2.4 Rekkefølge og prioritet

Å bruke Python som kalkulator er stort sett ganske rett frem. Derimot finnes det er par småting som kan være litt forvirrende for en nybegynner, så la oss se litt nærmere på disse. For å gjøre dette, la oss bruke et nytt eksempel.

La oss se på et nytt eksempel. I Fysikk lærer man de såkalte *bevegelsesligningene*, for eksempel har vi formelen

$$s = s_0 + v_0 t + \frac{1}{2} a t^2.$$

Her er s posisjonen til et objekt ved tiden t, dersom den starter i s_0 med farten v_0 og har konstant akselerasjon a.

Fra denne formelen lager vi en oppgave: Si at vi kaster en ball rett opp. I det ballen forlater hånden vår kan vi si at den er $s_0 = 1.5$ meter over bakken, beveger seg med en fart $v_0 = 12$ m/s og er kun påvirket av tyngdekraften $a = -9.81$ m/s^2.

Denne oppgaven kommer vi til å bruke som et gjennomgående eksempel i resten av dette kompendiet. For nå lurer vi bare på hvor høyt over bakken ballen er etter 1 sekund. Dette regner vi ut som følger:

```
s0 = 1.5 # m
v0 = 12  # m/s
a = -9.81 # m/s**2
t = 1 # s

s = s0 + v0*t + 0.5*a*t**2
print(s)
```

```
8.594999999999999
```

Igjen, merk at enhetene kun er skrevet på som kommentarer og påvirker ikke koden, vi legger dem bare på så det er lettere å holde oversikt over de ulike variablene.

2.4.1 Regnerekkefølge

Dette eksempelet er ikke så veldig forskjellig fra det forrige, men i dette tilfellet består formelen av to ledd. Man kan da lurte på å om Python respekteter riktig matematisk rekkefølge. Her kan du jo for eksempel gjøre den samme beregningen

for hånd for å sjekke, men svaret er at ja, Python bruker riktig rekkefølge. I dette tilfellet betyr det multiplikasjon før addisjon.

La oss gjør en annen kjapp test

```
x = 3 + 4/7
y = (3 + 4)/7
```

Her bør

$$x = 3 + \frac{4}{7} = \frac{25}{7} = 3.5714\ldots,$$
$$y = \frac{3+4}{7} = \frac{7}{7} = 1.$$

La oss kjøre koden vår å sjekke om Python får det samme:

```
print(x)
print(y)
```

```
3.571428571428571
1.0
```

Fra dette enkle eksempelet har vi ikke bevist at Python alltid bruker riktig regnerekkefølge, men det er altså tilfellet. Du kan også se at man fint kan bruke parenteser i utregningene sine ved å skrive dem rett inn i koden.

2.4.2 Koderekkefølge

Vi trenger altså ikke å ta spesielt hensyn til regnerekkefølge når vi koder, så lenge vi gjør det riktig matematisk sett har vi vårt på det tørre. Derimot kan det bli litt surr om vi ikke passer litt på rekkefølgen på kodelinjene våres.

Husk at Python tolker koden linje for linje, fra toppen ned. Dette innebærer at vi må definere variabelen som brukes i utregningen *før* vi gjør selve utregningen. Vi kan for eksempel ikke skrive

```
# (OBS: Dette kodeeksempelet vil _ikke_ fungere)
s = s0 + v0*t + 0.5*a*t**2

s0 = 1.5 # m
v0 = 12  # m/s
```

```
a = -9.81 # m/s^2
t = 1 # s

print(s)
```

Om vi skriver koden i denne rekkefølgen vil Python protestere med en feilmelding, fordi den ikke vet hva variablene `v0`, `a` og `t` er for noe.

Man kan lett tenke seg at denne koden også bør fungere, ettersom at man først definerer formelen, og så gir konkrete tallverdier. Slik oppgir man jo for eksempel ofte en matematikkoppgave. Men her er det viktig å tydeliggjøre at kodelinjen:

```
s = s0 + v0*t + 0.5*a*t**2
```

er en *utregning*, ikke en formel. Altså prøver Python å regne ut en konkret tallverdi når vi skriver denne linjen. Det finnes måter å definere formler på og, men da må vi bruke *funksjoner*, som vi skal lære om litt senere.

Når vi gjennomfører en utregning vil Python regne seg frem til én bestemt tallverdi som lagres i variabelen `s`, den vil altså ikke huske på *formelen* som produserte dette resultatet. En konsekvens av dette er at dersom vi endrer noen av parametrene `s0`, `v0`, `a` eller `t` *etter* utregningen, så vil dette *ikke* endre på `s`. Det kan du prøve selv.

2.5 Fler matematiske operasjoner

Python kommer med alle de grunnleggende artimetiske operasjonene "innebygd", slik at disse kan brukes fritt. Disse vises i Tabell 1. Ofte trenger vi matematiske funksjoner i tilegg til de vanlige operasjonene, for eksempel trigonometriske funksjoner eller logartimer. Disse er *ikke* innebygd i Python og må derfor *importeres* før vi kan bruke dem.

Si for eksempel at vi ønsker å bruke kvadratrot i en utregning. Dette kan vi gjøre med funksjonen `sqrt` (kort for square root). Da kan vi *importere* denne funksjonen fra et tileggsbiblotek på følgende måte:

```
from math import sqrt
```

Her sier vi til Python at vi ønsker funksjonen `sqrt` fra bibloteket `math`. Etter å kjørt denne kodelinja kan vi bruke funksjonen fritt i resten av koden vår. Det er

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`

Tabell 1: Merk spesielt at potens skrives `**`, og ikke `^` som i enkelte andre programmeringsspråk. Merk også at vi må skrive ut ab som `a*b`, da det ikke er implisitt multiplikasjon av variabler slik som i matematikk. Bruk av mellomrom mellom variabler og operatorer er også frivillig og påvirker ikke koden på noen måte, bruk derfor mellomrom slik at koden/formlene blir lettere å lese.

best å legge slike importeringer helt i toppen av programmet sitt.

La oss se på et helt program som importerer og bruker en funksjon. La oss bruke Pytagoras til å regne ut hypotenusen av en trekant:

```
from math import sqrt

a = 3
b = 4
c = sqrt(a**2 + b**2)

print(c)
```

```
5.0
```

Man kan lure på hvorfor man i det heletatt *trenger* å importere funksjoner, hvorfor er de bare ikke tilgjengelige hele tiden? Dette er egentlig et spørsmål om effektivitet. Python er et helt generelt programmeringsspråk, og brukes til langt mer enn realfag. Man kan for eksempel lage spill, nettsider, apper, osv. Det er derfor langt mer ryddig å ha funksjonalitet som man kun bruker i visse tilfeller i egne ekstrabiblotek, slik at selve Python kan være så liten, lettvekt og ryddig som mulig. Alle kan lage slike ekstrabibloteker fritt, så om man jobber på spesialiserte problemstilling, for eksmepel molekylærbiologi eller kvantefysiske beregninger, finnes det garantert egne pakker der ute som er laget for akkurat de temaene. Dette er også

en grunn til at vi anbefalte å laste ned Python via *Anaconda Distribution*, fordi da får du med tusenvis at ekstrabiblotek for realfag.

I eksempelet vårt importerte vi kun én funksjon (`sqrt`). Derimot kan man importere *alt* som er inneholdt i et biblotek ved å skrive asterisk/stjerne (`*`):

```
from math import *
```

Fordelen av dette er at man må skrive lite kode. Ulempen er at det ikke er like tydelig for nybegynnere hva som kommer fra hvor, om man importerer fra ulike biblotek.

Her vil vi også spesielt nevne bibloteket `pylab`. Denne pakken er spesielt laget for å gjøre det lett for de som ikke kan for mye programmering å jobbe med realfag. Det er derfor vanlig å skrive

```
from pylab import *
```

I toppen av slike program. Vi kan nevne at for alle eksempler vi vil bruke i dette kompendiet er `pylab` tilstrekkelig, så vi kommer til å bruke denne kommandoen en del.

Tabell 2 under lister vanlige matematiske funksjoner og hva de heter på "Pythonsk". Alle disse funksjonene finnes i `math`, men de ligger også i `pylab`, så de kan importeres fra begge steder.

I tilegg til funksjonene som er listes i Tabell 2 kan man 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
```

Matematiske funksjoner

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)`
Logaritme med base b	$\log_b(x)$	`log(x, b)`
Sinus	$\sin x$	`sin(x)`
Cosinus	$\cos x$	`cos(x)`
Tangens	$\tan x$	`tan(x)`
Sinus invers	$\sin^{-1} x$	`asin(x)`
Cosinus invers	$\cos^{-1} x$	`acos(x)`
Eksponentialfunksjonen	e^x	`exp(x)`
Fakultet	$n!$	`factorial(n)`

Tabell 2: Merk at disse funksjonene må importeres fra `math` eller `pylab` før bruk. De trigonometriske funksjonene er definert i radianer. Det er heller ingen konvensjon for å bruke ln for den naturlig logaritmen i informatikk, istedet brukes log for den naturlige logaritmen, og $\log_{10}$ for den briggiske.

2.6 Flette variabler inn i tekst

De fleste programmer vi lager i realfaglig sammenheng gjør en eller annen form for beregning. Siste steg vil derfor som oftest gå ut på å skrive ut resultatene slik at brukeren kan lese dem av. Her bruker vi gjerne `print`, men å kun skrive ut ett eller flere tall er ofte litt lite beskrivende, så vi ønsker gjerne og ha med litt beskrivende tekst og.

Vi har allerede kort sett et eksempel på hvordan vi kan skrive ut tekst og tall om hverandre med `print` ved å skille dem med komma. Ta for eksempel bevegelsesligningen over, her kan vi for eksempel skrive ut:

```
1 print("Etter", t, "sekunder er ballen", s, "meter over
    bakken")
```

```
Etter 1 sekunder er ballen 7.095 meter over bakken
```

Bruk av komma på denne måten er forholdvis intuitivt og fungerer bra. Derimot finnes det en enda bedre måte å gjøre det på. Den er litt mer vanskelig å lære seg,

men tilbyr langt bedre kontroll over utskriften.

Vi skal nå se på hvordan man kan kombinere tekst og variabler med såkalt *tekst-formattering*. Dette går ut på å "flette" variabeler inn i en tekststreng. La oss vise et enkelt eksempel:

```
navn = "Kari"
print(f"Hei på deg {navn}!")
```

```
Hei på deg Kari!
```

Her fletter vi navnet til personen inn i teksten, istedenfor å bruke komma. For å få til dette gjør vi to ting: Vi skriver en `f` rett før selve tekststrenger (dvs før apostrofen), og vi legger på krøllparenteser (`{}`) for å indikere at det er en variabel som skal flettes inn.

Vi skriver `f` foran teksten for å fortelle Python at tekststrengen skal formatteres, det vil si at datamaskinen da skjønner at `{navn}` refererer til en variabel, og den vil ta jobben med å flette variabelens innhold inn i teksten vår. Du kan huske at vi må skrive `f` foran strengen, fordi vi skal **f**lette inn variabler.

For eksempelet med bevegelsesligningen blir det da:

```
print(f"Etter {t} sekunder er ballen {s} meter over bakken
    ")
```

```
Etter 1 sekunder er ballen 7.095 meter over bakken
```

2.6.1 Finkontroll på print

En av fordelene ved å brukt print-formattering, eller "flette-printing" som vi kalte det er at vi kan ha finkontroll på utskriften vår. Det vanligste eksempelet er å kontrollere antall desimaler i utskriften vår. Si for eksempel at vi ønsker å holde oss til 1 desimal, da kan vi gjøre dette som følger:

```
from math import pi
R = 1.1
V = 4*pi*R**3/3
print(f"Ballen inneholder {volum:.1f} liter luft")
```

```
Ballen inneholder 5.6 liter luft
```

Her bruker vi formattering på vanlig måte, ved å skrive variabelen rett inn i teksten ved å bruke krøllparenteser ({}). Det vi har lagt til er at vi skriver `:.1f` i tilegg. Vi skriver først et kolon (`:`) for å signalisere at vi skal spesifisere *hvordan* variabelen skrives ut. Deretter skriver vi `.1f` fordi vi ønsker nøyaktig én desimal. Bokstaven `f` brukes fordi desimaltall gjerne kalles flyttall eller "floating" på engelsk.

Merk at når vi spesifiserer hvor mange desimaler vi ønsker, så runder Python av for oss, f.eks

```
from math import pi
print(f"{pi:.2f}")
print(f"{pi:.4f}")
```

```
3.14
3.1416
```

Ettersom at $\pi = 3.14159265\ldots$ blir π med 4 desimaler skrives ut til 3.1416.

Litt spesielt er også at Python ikke bruker den "vanlige" avrundingsregelen om at .5 alltid rundes opp. Men bruker istedet "bankers rounding". Litt interessant er det at Python først ble laget med "vanlig" avrunding, men dette endres med Python-versjon 3, ettersom at "bankers rounding" er vanligere i informatikkverden. Dette kan du lese mer om selv på for eksempel Wikipedia.

I tilegg til å påvirke antall desimaler kan vi også bruke formattering for å skrive ut i standardnotasjon, da bruker vi `e` istedenfor `f`. La oss ta et kjapt eksempel fra kjemi: Hvor mange molekyler H_2O er det i en liter med destillert vann?

For å regne ut dette regner vi først ut hvor mange mol en liter vann svarer til, og så ganger vi med Avogadro's konstant

```
m = 1000 # g
M = 18.015 # g/mol
avogadro = 6.022*10**23 # molekyler/mol

n = m/M # antall mol
N = n*avogadro
print(f"Det er {N:.1e} molekyler i 1 liter vann")
```

```
Det er 3.3e+25 molekyler i 1 liter vann
```

Her svarer `3.3e+25` til $3.3 \cdot 10^{23}$. Dette er en vanlig forkortet måte å skrive tall i standardnotasjon på som man ofte ser på kalkulatorer og lignende. Merk at vi også kan definere variabler på denne måten, f.eks:

```
avogadro = 6.022e23
```

Noen print-formatterings valg

Bokstav	Funksjon
`f`	Flyttall/desimaltall
`e`	Standardnotasjon
`g`	Desimaltall om tallet har liten eksponent, eller standardnotasjon
`%`	Skriv resultatet som prosent

Tabell 3

2.7 Overskrive og oppdatere Variabler

Vi har nå sett hvordan vi kan definere og regne med variabler. Det kan være fint å kjenne til at en variabel også kan endres i Python, de heter jo tross alt *variabler*.

Den enkleste måten å oppdatere en variabel er rett og slett å *redefinere* den, dette overskriver rett og slett det gamle innholdet.

```
masse = 0.5 # kg
masse = 2 # kg
print(masse)
```

```
2
```

Her erstattet vi rett og slett innholdet i variabelen fullstendig, og den første verdien blir glemt.

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 tilsvarende vis 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, og så videre.

2.7.1 Bruk av likhetstegn i kode

Et vanlig punkt for forvirring i programmering er bruk av likhetstegn. Dette er gjerne fordi man tenker på likhetstegnet i streng matematisk forstand. Når vi for eksempel skriver:

```
R = 2
A = pi*R**2
```

så ser dette veldig likt ut som man ville skrevet det matematisk. Men vi kan også for eksempel skrive:

```
x = x + 1
```

og dette er helt gyldig Pythonkode. Dette gir ikke mening matematisk, for en variabel kan jo ikke være lik seg selv pluss 1.

Forvirringen oppstår fordi likhetstegnet ikke er et matematisk symbol i Python, det betyr rett og slett at vi oppretter en variabel. Når vi skriver `=` i koden vår vil Python alltid se på høyresiden først, regne ut det som står der, dette blir variabelens verdi,

deretter tar den navnet på venstre siden. Dermed betyr kodelinjen `x = x + 1` at Python tar verdien til variabelen `x`, legger til 1, og overskriver den gamle verdien med den nye. Dette er altså tilsvarende som å skrive `x += 1`.

La oss sjekke dette:

```
x = 5
x = x + 1
print(x)
```

```
6
```

Det kan være lettere å tolke koden om man konsekvent tenker på likhetstegn som en pil som peker mot venstre, f.eks

```
x ← 5
x ← x + 1
print(x)
```

Merk at dette *ikke* er gyldig kode, men er bare ment for å illustrere en måte man kan tenke på det å definere variabler.

2.8 Brukerinteraksjon med input

Vi har nå sett hvordan vi kan definere variabler, regne med dem og skrive dem ut. Derimot kan det av og til være nyttig å lage programmer der konkrete variabler eller parametre ikke er satt i sten, men at de kan velges i det programmet kjøres. Senere i dette kompendiet skal vi for eksempel skrive et program som kan sjekke om et gitt tall er primtall eller ikke, og da kan det være fint å skrive programmet slik at man kan fylle inn tallet som skal sjekkes hver gang man kjører koden.

Det vi nå skal gjøre er å skrive kode som henter inn informasjon fra *brukeren* av programmet mens det kjører. Dette kalles gjerne *brukerinteraksjon*. Begrepet "bruker" her er kanskje litt rart, ettersom at vi ofte skriver og kjører våre egne programmer. Men det er et begrep som gjerne brukes for å tydeliggjøre forskjellene i rollen av å skrive kode, og bruke det endelige resultatet.

De fleste programmer du bruker på datamaskin og telefon har gjerne et *grafisk brukergrensesnitt* ("graphical user interface", GUI). Det er fullt mulig å lage slike

grensesnitt for våre egne programmer i Python, derimot tar det lang tid og kan være vanskelig. Derfor er det vanlig å rett og slett ikke benytte seg av grafiske grensesnitt i det heletatt når man programmerer vitenskapelig.

Derimot kan vi lage et langt enklere tekstbasert grensesnitt ved hjelp av funksjonen `input`. Denne funksjonen er veldig lik `print`, det vi bruker som funksjonsargument til `input` vil også skrives ut til konsollen på samme måte som print. Forskjellen kommer i at programmet vil pause opp og vente på et svar fra brukeren. Dette kan brukeren skrive inn i konsollen og først når de trykker "Enter" vil programmet fortsette.

La oss se på et enkelt eksempel:

```
1 navn = input('Hva heter du? ')
2 print(f'Hei {navn}. Hyggelig å møte deg!')
```

I denne koden definerer vi en variabel `navn`, men vi gir den ikke en konkret verdi. Istedet stiller vi brukeren et spørsmål ved hjelp av `input`. Det er svaret fra brukeren som blir til verdien av variabelen.

Om du kjører dette eksempelet selv vil du se at spørsmålet skrives ut til konsollen:

```
Hva heter du?
```

Derimot skjer det ikke så mye mer. Du kan nå trykke på konsollen og skrive inn ditt navn og avslutte med "Enter"-tasten. I eksempelkjøringer i dette kompendiet vil vi markere det brukeren skriver inn i blått:

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

Når vi skriver inn et navn sendes dette inn i koden, og siden vi skrev `navn = input(...)` lagres svaret i variabelen. Dermed skrives det ut en hyggelig, personlig beskjed.

Merk at vi må stille et spørsmål 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

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

for å ta vare på og huske på svaret fra brukeren.

2.8.1 Input og regning

Om vi ønsker å bruke `input` i programmene våres er det en liten ting vi må være obs på. Om det er slik at vi ønsker å spørre brukeren om input som skal tolkes som tall, så må vi være forsiktige at variabeltypene blir riktig.

Det er nemlig slik at Python vil alltid tolke `input` som en tekst. Selv om brukeren skriver inn et tall som svar, vil variabelen vi oppretter lagres som en tekststreng:

```
n = input("Velg et tall mellom 1 og 10: ")
print(type(n))
```

```
Velg et tall mellom 1 og 10: 7
<class 'str'>
```

Dette har konsekvenser om vi prøver å regne med tallet, fordi tekststrenger og tall behandles forskjellig.

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

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

```
7 + 8 = 15
```

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

Om vi derimot endrer hvordan vi definerer variablene til

```
a = '7'
b = '8'
```

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

```
7 + 8 = 78
```

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

På grunn av dette må vi rett og slett konverte input fra brukeren til tallverdier om vi ønsker å regne med dem. Dette gjør vi ved å skrive `int()` rundt input-funksjonen om vi ønsker heltall, eller `float()` rundt input-funksjonen om vi ønsker desimaltall. Så i eksempelet istad ville det blitt

```
1 n = int(input("Velg et tall mellom 1 og 10: "))
```

3 Tester og logikk

Vi skal nå begynne på et nytt grunnleggende programmeringskonsept, nemlig *logikk*. Når vi programmerer gir vi instrukser til datamaskinen for å løse et konkret problem i form av en algoritme. Derimot er det slik at for mange problemer må man gjøre forskjellige ting basert på omstendigheter. For eksempel kan et programmet gjør en beregning, og basert på utfallet, bestemme seg for om det må gjøre flere beregninger eller ikke.

For å innarbeide slik logikk i programmene våres må vi skrive *tester*. På Engelsk kalles slike tester gjerne for "if-tests". På norsk kalles de tilsvarende for "hvis-så" setninger, men dette er ikke like utbredt.

3.1 Tester i Python

For å skrive en test i Python bruker vi følgende kode:

```
if <betingelse>:
    <gjør noe>
```

Her må `<betingelse>` byttes med en betingelse som kan evalueres som enten sann eller falsk av datamaskinen. Tilsvarende må `<gjør noe>` erstattes av én eller flere kodelinjer som skal utøves *hvis, og bare hvis,* betingelsen er sann. Merk også at `in` er et nøkkelord vi må ha med, vi trenger også kolon (`:`) og *innrykket* før `<gjør noe>` kodelinjene.

En slik betingelse kan for eksempel være å sjekke at to variabler er like, eller kanskje at den ene er større enn den andre. La oss se et eksempel. I matematikk kaller vi en brøk som er større enn 1 for en *uekte* brøk. Den enkleste måten å se at en brøk er uekte er at telleren er større enn nevneren:

```
teller = 8
nevner = 5

if teller > nevner:
    print(f"{teller}/{nevner} er en uekte brøk")
```

```
8/5 er en uekte brøk
```

I denne koden skriver vi en betingelse som følger `teller > nevner`. Dette er en betingelse datamaskinen selv kan sjekke og evaluere til *sann* eller *falsk*. Vi kan faktisk skrive en slik betingelse uavhengig av tester også, selv om det ikke er like vanlig:

```
print(teller > nevner)
print(teller < nevner)
```

```
True
False
```

Ettersom at testen `if teller > nevner` evalueres til sann, så vil *innholdet* i testen, det vil si alle kodelinjer som har fått et lite innrykk, kjøres. Derfor ser vi at beskjeden skrives ut når vi kjører koden. Om testen *ikke* hadde vært sann, så ville Python glatt hoppet over all kode som hørte til testen. Dette kan du prøve selv, om du endrer `teller` til å være f.eks 3, så vil du ikke få skrevet ut noen beskjed.

Dette er et eksempel på en ren hvis-så setning: *hvis* en betingelse er gitt, *så* gjør vi noe. Om betingelsen ikke er sann gjør vi ingenting. I dette tilfellet derimot, kan det være nyttig å skrive ut en beskjed uansett, slik at brukeren forstår hva som foregår. Da kan vi istedet lage en *hvis-ellers* setning: *hvis* en betingelse er sann, så gjør en bestemt ting, og ellers, så gjør vi noe annet.

På engelsk skriver vi "else" for ellers, så da blir det som følger:

```
teller = 3
nevner = 5

if teller > nevner:
    print(f"{teller}/{nevner} er en uekte brøk")
else:
    print(f"{teller}/{nevner} er en ekte brøk")
```

```
3/5 er en ekte brøk
```

Her vil koden som hører til `if`-blokken kjøres om betingelsen er sann, ellers kjøres koden i *else*-blokken. Merk at *else*-blokken ikke trenger noen betingelse, fordi den inntreffer når den første betingelsen ikke gjør det.

La oss se på et annet eksempel. Vi sier at et tall er delelig med et annet dersom

det ikke er noen rest i divisjonen. Vi kan finne resten i en divisjon med *modulo*-operatoren, som skrives med % i Python

```
print(7 % 2)
print(8 % 5)
print(12 % 4)
```

```
1
3
0
```

Så vi kan sjekke om et tall er delelig med et annet ved å sjekke om resten er 0. For å sjekke om et tall er lik et annet bruker vi to likhetstegn (==), dette er fordi ett likhetstegn brukes til å definere variabler, så vi bruker to for å være tydelige på at vi snakker om faktisk matematisk likhet.

```
a = 131034
b = 3

if a % b == 0:
    print(f"{a} er delelig med {b}.")
else:
    print(f"{a} er ikke delelig med {b}.")
```

```
131034 er delelig med 3
```

Selv om modulo-operatoren kanskje er ny for noen er dette eksempelet ganske enkelt, men vi kommer til å bruke det til å finne primtall når vi kommer til kapittelet om sammensatte eksempler.

Tabell 4 viser de vanligste operasjonene vi bruker når vi sammenligner variabler.

3.2 Flere utfall

La oss se på et litt mer interessant eksempel. Når vi løser andregradsligninger med abc-formelen har vi tre mulige utfall, en gitt ligning kan ha to, ett, eller ingen nullpunkter. La oss skrive en test som kan behandle hver av de tre tilfellene separat.

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

Tabell 4: 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 sorterterte dem alfabetisk.

Når vi har fler enn to valg kan vi ha en `if`-blokk og en `else`-blokk som vanlig, men vi kan ikke ha mer enn en slik. Flere valg legger vi til iform av `elif`-blokker, dette raret ordet står rett og slett for "else-if".

Måten man vet hvor mange nullpunkter ligningen har er ved hjelp av diskriminanten, så gitt at vi har a, b og c kan vi regne ut denne og sjekke om den er større, lik, eller mindre enn 0.

```
a = 2
b = 2
c = -4

diskriminant = b**2 - 4*a*c

if diskriminant > 0:
    print(f"Ligningen har to røtter")
elif diskriminant == 0:
    print(f"Ligningen har 1 rot")
else:
    print(f"Ligningen har ingen røtter")
```

I dette eksempelet har vi først en `if`-test med én betingelse, så en `elif` med en annen betingelse, og til slutt en `else` som fanger opp alle tilfeller som ikke ble fanget opp av de to første.

Merk at vi kan ha så mange valg vi måtte ønske. Si for eksempel at vi driver å regner ut spektrallinjer i Fysikken, da ønsker vi kanskje å oversette en bølgelengde i nanometer til en viss farge. Dette kan vi gjøre med en litt lang if-test som følger:

```
bølgelengde = 656.3 # nanometer

if λ < 380:
    farge = "Ultraviolett"
elif λ < 450:
    farge = "Fiolett"
elif λ < 485:
    farge = "Blå"
elif λ < 500:
    farge = "Turkis"
elif λ < 560:
    farge = "Grønn"
elif λ < 590:
    farge = "Gul"
elif λ < 625:
    farge = "Oransj"
elif λ < 740:
    farge = "Rød"
else:
    farge = "Infrafrødt"
```

```
Lys med bølgelengde på 656.3 nm er rødt
```

Merk at vi her velger å bare sjekke *mindre enn* biten ved hver test. Istedenfor å f.eks sjekke $380 < \lambda < 450$. Grunnen til dette er at dersom én blokk slår inn, så sjekker ikke Python de resterende. Det vil si at dersom $\lambda < 380$ så er vi ferdig. Om vi går videre trenger vi altså ikke å sjekke nedover, vi kan anta at bølgelengden er lengre. Til slutt kommer `else`-blokken, hvor vi da vet at $\lambda \geq 740$.

Merk at vi her har valgt å bruke λ som variabelnavn. Dette fungerer fint i Python, men vi må isåfall klippe den inn fra et annet program om man ikke klarer å skrive denne på tastatur. Alternativt kan man bare skrive ut "bølgelengde" eller lignende.

3.2.1 Boolsk Logikk

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

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

4 Løkker

Løkker brukes for å gjenta en prosess mange ganger. Dette høres kanskje ikke så nyttig ut, men er det av de viktigste konseptene innen programmering. For det første kan de spare oss mye arbeid ved at vi kan skrive en kort kode som datamaskinen gjennomfører mange ganger. En langt viktigere grunn derimot er at veldig mange *algoritmer* laged med løkker, det vil si at løkker lar oss løse viktige problemstillinger. For eksempel er numerisk derivasjon, integrasjon og løsning av differensialligninger alle problemstillinger som krever løkker.

Det er vanlig å snakke om to forskjellige type løkker i programmering, og disse kalles gjerne for *for*-løkker og *while*-løkker på engelsk. I bunn og grunn er de to veldig like, og vi kommer tilbake til forskjellene når vi har gått igjennom dem hver for seg.

4.1 For-løkker

En for-løkke gjentar en bit med kode for hvert element i en *sekvens*, derav navnet. La oss vise hva vi mener med et eksempel.

I ballkast-eksempelet vårt fra tidligere regnet vi ut formelen

$$s = s_0 + v_0 t + \frac{1}{2} a t^2.$$

Når vi regnet på denne ligningen gjorde vi derimot dette kun for én bestemt tid. Ofte ønsker man jo å vite hvordan ballen flytter seg over tid. La oss derfor bruke en løkke til å regne ut s for en rekke ulike tidspunkt, altså avstanden som funksjon av tid $s(t)$. Dette kan vi gjør som følger:

```
s0 = 1.5
v0 = 12
a = -9.81

for t in [0, 0.5, 1, 1.5, 2, 2.5]:
    s = s0 + v0*t + 0.5*a*t**2
    print(f"{t:.1f}   {s:.1f}")
```

```
0.0   1.5
0.5   6.3
1.0   8.6
```

```
1.5    8.5
2.0    5.9
2.5    0.8
```

Denne koden introduserer en del nytt, så la oss ta det steg for steg.

La oss se på selve løkka først. Vi skriver `for t in [...]:`. Her bruker vi nøkkelordene `for` og `in` til å spesifisere at vi skal lage en nøkkel. Vi skriver også variabelnavnet `t`, merk at denne ikke er definert tidligere i koden. Vi kaller dette for en *løkkevariabel*, ettersom at denne variabelen vil endre seg for hver gang løkka gjentar seg. Til slutt må vi spesifisere hva vi skal løkke igjennom. Her spesifiserer vi en *liste*. Vi skriver en rekke tall, separert med komma, og omringer dem med firkantparenteser (`[]`). Her sier vi rett og slett at vi skal gjenta en bit med kode flere ganger, der først $t = 0$, så $t = 0.5$, så $t = 1.0$ osv. Merk til slutt bruk av kolon (`:`), slik som ved if-tester. Dette kolonet er vanlig å glemme.

Etter selve løkka kommer to kodelinjer som er gitt innrykk. Dette betyr at disse kodelinjene hører til løkka, og det er denne koden som vil gjentas hver gang løkka repeterer. Det vi gjør på disse to linjene er å regne ut s på nytt, for hver nye t verdi, deretter skriver vi ut resultatet.

I utskriften ser vi at løkka har gjentatt seg selv som forventet. For hver gang løkka repeterer har vi en ny `print`-kommando, som gir en ny linje med utskrift. Resultatet blir en tabell, der vi kan se en rekke verdier av s for forskjellige tider.

La oss se på et annet eksempel. Si vi setter 10 000 kroner på en høyrentekonto. Hvordan vil pengene vokse over de neste 5 årene? Dette kan vi enkelt finne ved hjelp av en for-løkke

```
1 penger = 10000
2 rente = 1.0345 # Tilsvarer rente på 3.45% p.a.
3
4 for år in [1, 2, 3, 4, 5]:
5     penger *= rente
6     print(f"Etter {år} år er saldo {penger:.0f} kr.")
```

```
Etter 1 år er saldo 10345 kr.
Etter 2 år er saldo 10702 kr.
Etter 3 år er saldo 11071 kr.
Etter 4 år er saldo 11453 kr.
Etter 5 år er saldo 11848 kr.
```

I dette eksempelet gjentar vi en rentejustering hver gang løkka repeterer (hvert år), dette gjør vi ved å skrive `penger *=rente`. Dette betydde å gange verdien i en variabel med et tall, i dette tallet 1.0345.

I begge disse eksemplene har vi løkket over lister av tall. Det går også fint å løkke over lister av andre typer variabler, for eksempel en liste med strenger.

I begge eksemplene så langt har vi løkket over lister av tall. Det går også fint å løkke over andre typer sekvenser. For eksempel kan man løkke over en tekststreng, da går vi isåfall igjennom hver karakter i strengen. Et bruksområde for dette er for eksempel for å gå igjennom å behandle en DNA-sekvens for eksempel.

4.1.1 Eksempel: DNA-sekvenser

DNA består av en lang rekke *nukleotider* med fire forskjellige nitrogenbaser. Vi kan derfor skrive en DNA-sekvens som en tekststreng bestående av bokstavene A, T, C og G.

```
sekvens = "TAGTGTGCGGGGAACGAGGCTTCTTCTACA"
```

Det er en lang rekke med analyse og endringer vi kan gjøre til en slik sekvens, og mange av dem vil benytte seg av løkker. La oss se på ett slikt eksempel.

DNA er dobbeltrådet, der A i den ene tråden alltid er parret med T, og C alltid med G. De to trådene kalles blant annet for malstrengen og kodingstrengen. Si at vi har en sekvens oppgitt slik den ser ut langs malstrengen og ønsker å finne den tilsvarende kodingstrengen. Da må vi altså oversette tråden. Det kan vi gjøre slik:

```
malstreng = "TAGTGTGCGGGGAACGAGGCTTCTTCTACA"
kodestreng = ""

for base in malstreng:
    if base == "T":
        kodestreng += "A"
    elif base == "A":
        kodestreng += "T"
    elif base == "C":
        kodestreng += "G"
    elif base == "G":
        kodestreng += "C"
```

```
    else:
        print("Ugyldig karakter funnet i DNA-streng!")

print(kodestreng)
```

```
ATCACACGCCCCTTGCTCCGAAGAAGATGT
```

Merk at vi legger til en `else`-blokk i testen vår. Denne vil kun slå inn om det finnes en karakter i DNA-strengen vår som *ikke* er en av de 4 basene (A/T/G/C). Dette legger vi inn som en slags idiotsikring, slik at vi får en beskjed om noe ser ut til å ha blitt feil. Vi har nå skrevet kode som kan oversette enhver malstreng til en kodingstreng eller motstatt. Nå kan du kanskje lure på hva vitsen med dette er, og det kommer vi tilbake til senere i kurset når vi skal se litt nærmere på hvordan man kan generere proteiner fra DNA-sekvenser.

4.2 Løkke over tallrekker med range

Selv om det er fullt mulig å løkke andre ting enn tallrekker, er det i praktisk oftest nettopp tallrekker vi er interessert i. Derimot har vi så langt skrevet ut lister av tall vi skal løkke over manuelt. Dette fungerer forsåvidt greit når det er snakk om et få antall tall, men det skalerer fryktelig dårlig. Si for eksempel vi ønsker å løkke over alle tall fra 1 opp til 1000? Dette er det ikke praktisk gjennomførbart å skrive ut selv. På den andre siden er det også vanskelig å generalisere, si vi ønsker å løkke over tallene $1, 2, \ldots n$, der n er en variabel som kan endre seg for hver gang vi kjører programmet vårt. Hvordan får vi til det?

For å automatisere prosessen av å løkke over en tallrekke kommer Python med en nyttig funksjon: `range`. "Range" betyr tallrekke på engelsk og er en funksjon som lar oss spesifisere en tallrekke på en kort å konsis måte.

Man kan bruke *range* med ett, to eller tre argumenter. La oss ta for oss alle tre tilfellene:

Ett argument) Om vi bare gir ett argument til funksjonen, dvs, `range(n)` får vi tallrekka

$$0, 1, \ldots, n-1,$$

altså tallene fra og med 0 opp til, men ikke med, n. Grunnen til at vi begynner på 0 og ikke 1 er at det er en vanlig konvensjon i informatikk å starte å telle ved 0.

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

To argumenter) Om vi gir to argumenter til funksjonen, dvs `range(a, b)` får vi tallrekka fra og med a, til, men ikke med, b, altså $[a, b)$.

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

```
2
3
4
```

Tre argumenter) Om vi legger på et tredje argument kan vi øke med mer enn ett steg av gangen om ønskelig. Så med tre argumenter kaller vi dem ofte for: vi `range(start, stop, step)`.

```
for i in range(-4, 6, 2):
    print(i)
```

```
-4
-2
0
2
4
```

Merk at 6 ikke kommer med, fordi range alltid går opp til, men ikke med, slutt-verdien.

Med hjelp fra range kan vi enkelt løkke over tallrekker. F.eks kan vi endre rente-beregningen over til

```
for år in range(1, 21):
    penger *= rente
    print(f"Etter {år} år er saldo {penger:.0f} kr.")
```

For å se på saldoen over de neste 20 årene.

For eksempelet med $s(t)$ ser vi derimot fort en begrensning i `range`, og det er at den kun jobber med heltall. Altså kan vi ikke enkelt løkke over desimtallverdier av t. Derimot kan funksjonen `arange` fra pylab gjøre dette for oss.

```
from pylab import *

v0 = 12
a = -9.81

for t in arange(0, 3, 0.1):
    s = v0*t + 0.5*a*t**2
    print(f"{t:.1f}   {s:.1f}")
```

Her vil løkka gå fra $t = 0$ til $t = 3$ i steg på 0.1. Merk at $t = 3.0$ ikke vil regnes ut, slik som tidligere.

4.3 Eksempler: Sum av tallrekker

Ett eksempel der bruk av løkker passer godt er når vi skal regne med tallrekker i matematikken, for eksempel for å regne ut summen av en slik rekke. La oss se på hvordan vi kan gjøre dette i Python.

Vi begynner med muligens den enkleste summen vi kan finne, nemlig

$$S_n = 1 + 2 + 3 + \ldots n.$$

Her kalles gjerne S_n for det n-te trekanttallet. Vi kan bruke en for-løkke og `range` til å løkke over disse tallene

```
for i in range(1, n+1):
```

For å faktisk finne summen må vi opprette en *summevariabel*, der vi kan legge til hvert enkelt bidrag etterhvert som vi løkker over det. La oss velge $n = 100$ bare for å illustrere

```python
n = 100
S = 0

for i in range(1, n+1):
    S += i

print(S)
```

```
5050
```

Merk at vi her har valgt å kalle summevariabelen `S`. Dette velger vi fordi koden da ligner mer på det matematiske uttrykket vi har skrevet. Vi kan alternativt velge et mer beskrivende variablenavn som f.eks `total`. Merk at `sum` er en innebygd funksjon i Python, så vi bør ligge unna dette navnet.

Det er lett å gjøre feil når man programmerer, så dette er en perfekt mulighet til å sammenligne det *numeriske* svaret vi har funnet, med det vi finner med en matematisk formel. Om man har lært om tallrekker og notasjon av summer kan summen skrives mer kompakt som

$$S_n = \sum_{i=1}^{n} i.$$

Merk at dette uttrykket ligger enda nærmere koden vår. Artimetiske summer av denne typen finnes det generelle formler for å løse, så om man slår opp disse finner vi at

$$S_n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Om vi kobler inn 100 får vi 5050, samme svar som med koden vår. Dette tyder på at vi har fortstått problemet, og skrevet riktig og gyldig kode.

Ettersom at vi har en formel for de aritmetiske rekkene er det strengt tatt ikke nødvendig å programmere dem. La oss derfor se på en litt mer komplisert rekke, for eksempel:

$$S = \sum_{k=1}^{30} \frac{k^2+1}{k^2+2k+1}.$$

Matematisk ser dette med en gang mye verre ut. Dette ligner ikke på en standard rekke man lærer seg på VGS, og om man blar litt i formelsamlinger finner man ingenting som ligner. Her kan vi kanskje faktorisere noen uttrykk eller forenkle

på noen måte, eller så kan det hende at faktisk ikke finnes noe lukket uttrykk for denne rekka i det heletatt?

Med vår numeriske fremgangsmåte i Python derimot, er denne rekka nesten helt lik som før, vi bare endrer selve innholdet i løkka:

```
S = 0
for k in range(1, 31):
    S += (k**2+1)/(k**2 + 2*k + 1)
print(S)
```

```
25.17189100978281
```

Fremgangsmåten vår i Python er altså langt mer generell enn de vi gjør med penn og papir, som er spesialisert til rekker av helt bestemte former og mønstre. Vår numeriske fremgangsmåte kan brukes til å finne summen av så og si hvilken som helst endelig sum, og metoden kan faktisk brukes til å finne summen av en del konvergente uendelige summer også!

Selv om eksempelet her kanskje føles litt for enkelt illustrer det en av de store styrkene til numeriske metoder. For derivasjon, integrasjon og løsing av differensialligninger kan alle uttrykkes som summer av endelige tallrekker og løses på tilsvarende vis. Dette kommer vi tilbake senere i kurset når vi skal se på Euler's metode for å løse differensialligninger i Fysikken.

4.3.1 Eksempel: Fakultet

La oss se på et eksempel som ligner veldig på sum av tallrekke, nemlig produkt av tallrekke. Fakultet er en viktig matematisk operasjon som blant annet brukes mye i kom binatorikk. Vi skriver fakultet med et utropstegn i matematikken. For eksempel er

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

Altså ganger vi tallrekka nedover til og med 1.

Vi kan lage et kort program som regner ut $n!$ for oss, ved hjelp av en løkke som følger:

```
n = 5
total = 1
for tall in range(1, n+1):
    total *= tall
print(f'{n}! = {total}')
```

```
5! = 120
```

Her testet vi med 5! og så at svaret ser rimelig ut. Her kan vi endre n og sjekke forskjellige verdier for å være mer sikre.

Om vi ønsker å gå et steg lenger kan vi bruke en *dobbel løkke* til for eksempel å regne ut $n!$ for $n = 1, \ldots 5$.

```
for n in range(1, 6):
    total = 1
    for tall in range(1, n+1):
        total *= tall
    print(f'{n}! = {total}')
```

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

Slike doble løkker kan fort bli uoversiktlig, så her er det viktig å holde tunga rett i munnen. Det er spesielt innrykkene det er viktig å følge med på her. Merk at kodelinjene 2–5 alle har innrykk, dermed vil de alle gjentas for hver gang n oppdateres. Ettersom at linje 3 definerer en ny løkke lages det en ny løkke hver gang n oppdateres. Merk linje 4 spesielt, den har fått dobbelt innrykk ettersom at den hører til den innerste løkka.

Ta gjerne nok tid til å analysere dette eksempelet for å forstå det skikkelig. Som vanlig kan det være en god idé å gå frem kodelinje for kodelinje og "simulere" datamaskinen selv. Ta gjerne penn og papir og skriv ned hva de ulike variablene er til enhver tid etterhvert som du går igjennom koden.

Vi kan nå øke hvor mange tall vi skal inkludere for å få en forståelse for hvor raskt fakultet egentlig vokser. Den vokser faktisk raskere enn eksponentielt!

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

```
n = 52
total = 1
for tall in range(1, n+1):
    total *= tall
print(f'{n}! = {total}')
```

```
52! = 80658175170943878571660...
      63685640376697528950544...
      088327782400000000000
```

et helt enormt tall! La oss endre print-fortmatteringen til `total:g` slik at vi får standardnotasjon på store tall. Da får vi

```
52! = 8.06582e+67
```

Som svarer til 8×10^{67}. Det er et så stort tall at det er nesten umulig å forklare det. Hver gang en kortstokk stokkes (skikkelig) velges én av disse permutasjonene tilfeldig. Det betyr at vi med sikkerhet kan si at når man stokker en kortstokk skikkelig, vil den ha en rekkefølge som er helt unik, sammenlignet med alle andre stokkinger som har vært i historien.

4.4 While-løkker

La oss nå snu oppmerksomheten mot den andre typen løkker vi har tilgjengelig, nemlig *while*-løkker. La oss begynne med det samme eksempelet som vi brukte med for-løkkene, ballkastet:

$$s(t) = s_0 + v_0 t + \frac{1}{2} a t^2.$$

Vi har allerede vist hvordan vi kan bruke en for-løkke til å regne ut denne formelen for en rekke ulike verdier av t. Derimot må vi med en for-løkke på forhånd bestemme oss for hvilke verdier av t vi skulle bruke. Men si at vi isteder ønsker å fortsette helt til ballen treffer bakken, dvs, til $s = 0$? Da måtte vi rett og slett prøvet oss frem til vi fant fornuftige verdier av t.

Med en while-løkke derimot, så kan vi eksplisitt si at vi skal fortsette å regne oss fremover i tid, helt til objektet treffer bakken. Måten vi gjør dette på ligner litt på en if-test, ved at vi bruker en betingelse. La oss se på koden:

```
# Parametre
s0 = 1.5
v0 = 12
a = -9.81

# Tidssteg
dt = 0.1

t = 0
s = s0

while s >= 0:
    t += dt
    s = s0 + v0*t + 0.5*a*t**2
    print(f"{t:2.1f}  {s:4.2f}")
```

Her setter vi først de vanlige parametrene `s0`, `v0` og `a`. Men nå har vi også lagt til en variabel vi kaller `dt`, som står for Δt. Dette er *tidssteget* vårt, som rett og slett er hvor langt fremover i tid vi skal gå for hvert steg i utregningen vår. Deretter sier vi at vi skal starte på $t = 0$ og $s = s_0$. Så kommer selve løkka, hvor vi skriver `while s >=0:`.

Her ligner en while-løkke mye på en if-test, i det at vi skriver en betingelse og så forgrener koden seg avhengig av om betingelsen er sann eller ikke. Forskjellen fra if-testen er at dersom betingelsen er sann og kode kjøres, så sjekkes betingelsen på nytt. Altså gjentar while-løkka seg gang på gang, helt til betingelsen ikke lenger er sann. Det er her navnet kommer fra, da `while s >=0:` kan leses som "mens ballen er over bakken, fortsett".

Inne i løkka må vi eksplisitt oppdatere tiden `t`, noe vi ikke trengte i for-løkka. Etter `t` er oppdatert regner vi ut `s` på nytt og skriver ut resultatet. Utskriftene fortsetter helt til ballen treffer bakken

```
0.1  2.65
0.2  3.70
...
2.5  0.84
2.6  -0.46
```

Her ser vi altså at while-løkka går opp til $t = 2.6$ og stopper der av seg selv, ettersom at da blir s negativ, og betingelsen blir dermed falsk. Merk at vi får skrevet ut én verdi av s som er negativ—klarer du å forklare hvorfor?

Si at vi ønsker å finne nøyaktig den tiden ballen treffer bakken. Vi ser fra utskriften over at det skjer et sted mellom 2.5 og 2.6 sekunder. Men vi klarer ikke å si akkurat hvor innenfor dette intervallet det skjer. Om vi minsker tidssteget vårt derimot, får vi en bedre "oppløsning". Når vi gjør dette vil vi også automatisk få langt fler utskrifter, så vi velger å flytte `print`-funksjonen utenfor løkka, så vi kun får skrevet ut den siste verdien:

```
dt = 0.001
while s >= 0:
    t += dt
    s = s0 + v0*t + 0.5*a*t**2
print(f"{t:2.3f}  {s:4.3f}")
```

```
2.566  -0.004
```

n Her får vi altså skrevet ut den første tiden der ballen har truffet bakken. Igjen kan denne verdien finnes for hånd ved å løse ligningen

$$s(t) = s_0 + v_0 t + \frac{1}{2} a t^2 = 0.$$

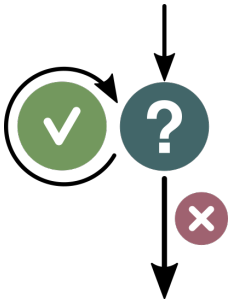
Dette kan gi en god kontroll på at koden vår fungerer som forventet.

Man kan lure på hvorfor det er nyttig å finne tiden ballen treffer bakken ved hjelp av en løkke når det enkelt kan gjøres for hånd. Men senere i kurset skal vi se hvordan vi kan legge til luftmotstand i modellen vår, og da klarer man ikke lenger å finne denne tiden for hånd. Den numeriske løsningen vår derimot, vil fortsatt fungere.

4.5 Uendelige løkker

Vi har nå sett både for-løkker og while-løkker. I for-løkkene må vi spesifisere hvor langt løkka skal gå i det vi lager den. I while-løkkene derimot, bestemmes dette av betingelsen vi setter. Situasjonen kan tegnes som i Figur 3.

En feil det er lett å gjøre når man lager en while-løkke, er rett og slett å lage en uendelig løkke ved uhell. Ta for eksempel ballkastet vi gjorde istad. Her kan man få problemr om man glemmer å oppdatere tiden i løkka, slik som dette:



Figur 3: En while-løkke minner om en if-test, med den viktige forskjellen at der en if-test kun kjører én gang vil en while-løkke gjenta seg selv på nytt og på nytt så lenge betingelsen fortsatt er sann.

```
while s >= 0:
    s = s0 + v0*t + 0.5*a*t**2
    print(f"{t:2.3f}  {s:4.3f}")
```

Her har vi glemt linja `t += dt`. Her regner vi ut s "på nytt" hver gang løkka repeterer, men siden t ikke endrer seg vil s bli den samme hver gang. Vi vil altså få følgende utksrift

```
0.000  1.500
0.000  1.500
0.000  1.500
0.000  1.500
...
```

Og denne bare fortsetter og fortsetter. Vi har rett og slett laget en uendelig løkke.

Når vi lager en slik uendelig løkke vil ikke Python automatisk stoppe og programmet vil fortsette å kjøre "for alltid". Datamaskinen antar rett og slett at vi vet hva vi driver med og gjør akkurat det vi spør om.

Om du lager en slik uendelig løkke ved uhell, og det skjer absolutt alle som programmerer nok, så må du avbryte kjøringen av programmet manuelt. I Spyder gjør du dette ved å trykke `Stopp`-symbolet som er en rød firkant over outputkonsollen. Merk at den blå firkanten i verktøylinja til høyre for selve kjør-knappen er til noe helt annet, og vil ikke avbryte en uendelig løkke. Etter man har avbrutt kjøringen kan man fikse på løkka og prøve igjen.

La oss understreke at det ikke er noe farlig å lage en uendelig løkke, så du trenger ikke være redd for å prøve deg frem. Det er bare litt irriterende å ha en løkke kjørende i Spyder, for du får ikke kjørt et nytt program før du stopper det forrige.

4.5.1 Uendlige løkker med vilje

Selv om det kanskje høres rart ut ønsker man av og til å lage en uendlig løkke med vilje. Dette kan man gjøre ved å skrive

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

Her vil betingelsensjekkes hver gang og være sann, slik at koden bare gjentar og gjentar seg selv. Vi kan inne i koden avbryte en slik løkke med `break` som bryter ut av en løkke (også for-løkker). T

Ta for eksempel dette "kalkulator"-programmet som regner ut fakultet av tallene brukeren gir helt til de skriver `slutt`.

```
while True:
    svar = input("Gi n (eller 'slutt' for å avslutte): ")
    if svar == "slutt":
        print("Programmet avslutter.")
        break
    else:
        n = int(svar)
        total = 1
        for i in range(1, n+1):
            total *= i
        return total
```

```
Gi n (eller 'slutt' for å avslutte): 5
5! = 120
Gi n (eller 'slutt' for å avslutte): 8
8! = 40320
Gi n (eller 'slutt' for å avslutte): 12
12! = 479001600
Gi n (eller 'slutt' for å avslutte): slutt
Programmet avslutter.
```

4.6 For- eller while-løkke?

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

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

Om man jobber med en konkret problemstilling i klasserommet er det ofte ikke nødvendig å vite om begge type løkker engang, om du derfor jobber med elever som har liten til ingen programmeringserfaring kan man unnlate å nevne den andre typen løkken for å holde ting så enkelt som mulig.

Det er også debatt om hvilke løkker det er lurest å lære bort først, og dette handler også om hva man syns er mest intuitivt. Noen mener at *while*-løkker er mer intuitive fordi de springer naturlig ut av if-testene. Andre mener at while-løkker blir mer kompliserte fordi de avhenger av hva som skjer når koden faktisk kjører for å forstå når de avslutter. Her er det ingen fasitsvar. Det som derimot *er* lurt, er å dekke if-tester før while-løkker.

4.6.1 Eksempel: Høyrentekonto

Vi så tidligere på et eksempel med en høyrentekonto. Dette gjorde vi originalt med en for-løkke, og dette er den beste løsningen om vi vet hvor mange år fremover vi er interessert i å regne ut. Om vi derimot snur problemet på hodet og spør: *hvor mange år må vi vente før pengene på konto har doblet seg?*, så er while-løkker det bedre valget.

```
1 penger = 10000
2 rente = 1.0345
3 år = 0
4
5 while penger < 20000:
6     penger *= rente
7     år += 1
8
```

```
9  print(f"Du må vente {år} før pengene har doblet seg")
10 print(f"Det er da {penger} kroner på konto.")
```

```
Du må vente 21 før pengene har doblet seg
Det er da 20386 kroner på konto.
```

5 Plotting

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

For å plotte skal vi bruke et tilleggsbiblotek som heter Matplotlib. Dette har blitt installert automatisk som en del av Anaconda-pakken, og importeres sammen med *pylab*. Det holder altså å skrive:

```
from pylab import *
```

På denne måten har vi tilgang på alt vi trenger for å plotte i programmene våres.

5.1 Kurveplott

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

La oss gå tilbake til ballkast-eksempelet vårt. La oss prøve å plotte $s(t)$ med tiden langs x-aksen. Vi har allerede sett hvordan vi kan regne ut en rekke slike verdier ved hjelp av en løkke, men nå må vi lære hvordan vi kan lagre disse verdiene mens vi løkker.

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

```
s0 = 1.5
v0 = 12
a = -9.81

dt = 0.1
```

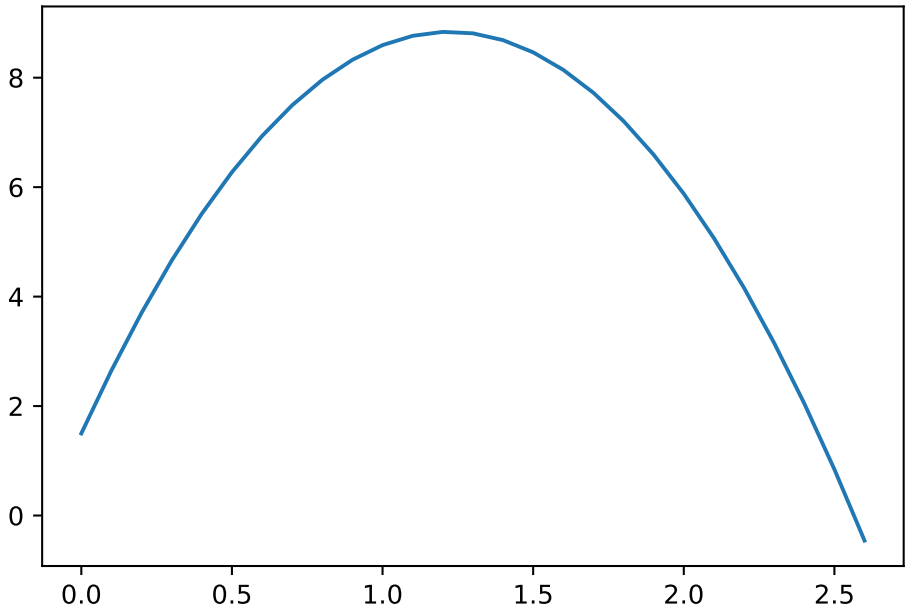
```
6  t = 0
7  s = s0
8
9  t_verdier = [t]
10 s_verdier = [s]
11
12 while s >= 0:
13     t += dt
14     s = s0 + v0*t + 0.5*a*t**2
15
16     t_verdier.append(t)
17     s_verdier.append(s)
```

Her har vi nøyaktig samme while-løkke som har brukt tidligere, vi har bare lagt til linje 8 og 9, som definerer to listevariabler, og linje 15 og 16, som bygger opp disse listene element for element. Før lista kjører inneholder listene kun $t = 0$ og $s = s_0$. Men hver gang løkka repeterer seg legger vi til en ny t-verdi og en ny s-verdi. Før vi går videre kan du prøve å printe ut listene for å se hvordan de ser ut.

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

```
1 from pylab import *
2
3 # Fyll inn kode for å finne t- og s-verdiene
4
5 plot(t_verdier, s_verdier)
6 show()
```

Figuren vi får ut ser ut som følger:

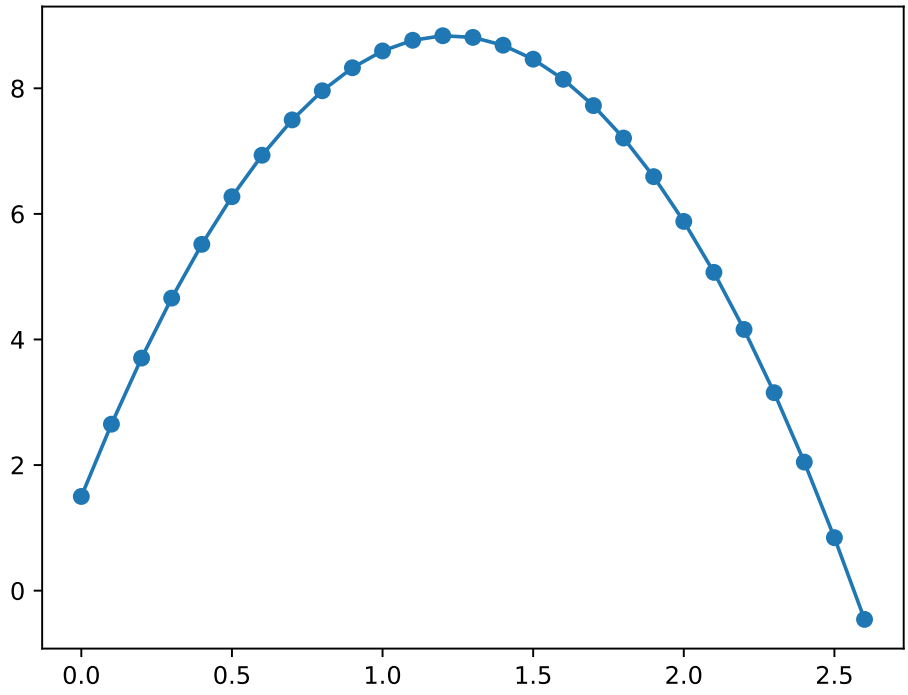


Figur 4: Ballens høyde over bakken som funksjon over tid. Denne figuren er det vi får når vi kun bruker `plot()`, uten noe ekstra "pynt".

Denne kurven ser buet og jevn ut, men den er egentlig bare en rekke rette linjer som knytter sammen datapunktene vi fant i løkka vår. Vi kan vise dette tydeligere ved å legge til en liten `'o-'` til `plott`-argumentet vårt:

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

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



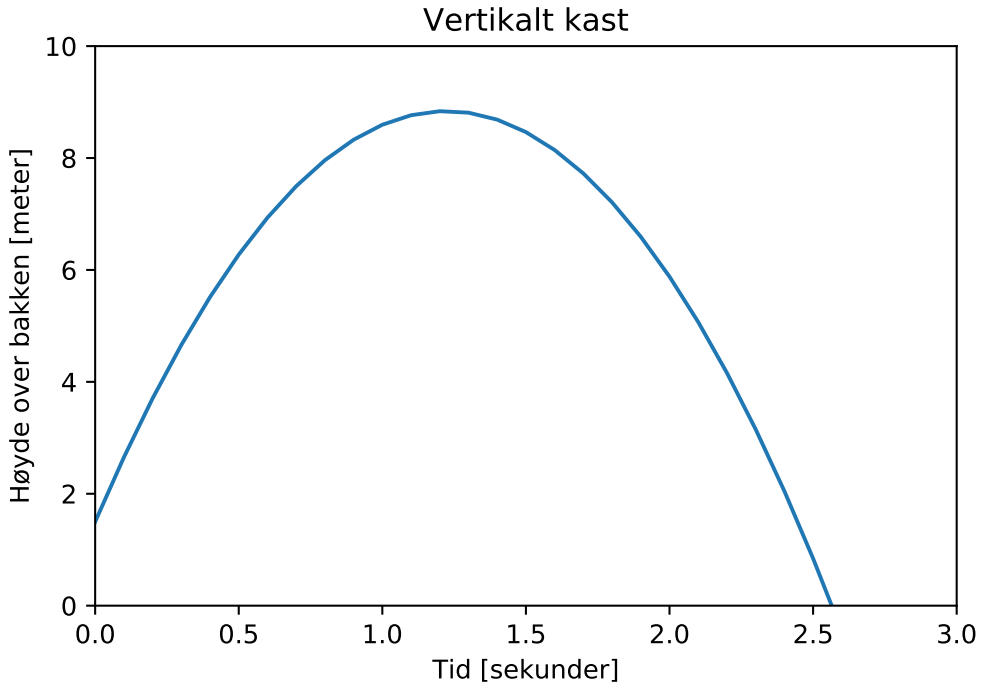
Figur 5: Ballens høyde over bakken som funksjon over tid når vi eksplisitt viser frem hvert datapunkt.

5.1.1 Pynte på plott

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

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

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



Figur 6: Ballens høyde over bakken som funksjon over tid. Plottet er nå blitt "pyntet" på med navn på akser og et manuelt valgt vindu.

5.1.2 Eksempel: Plotte andregradsfunksjoner

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

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

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

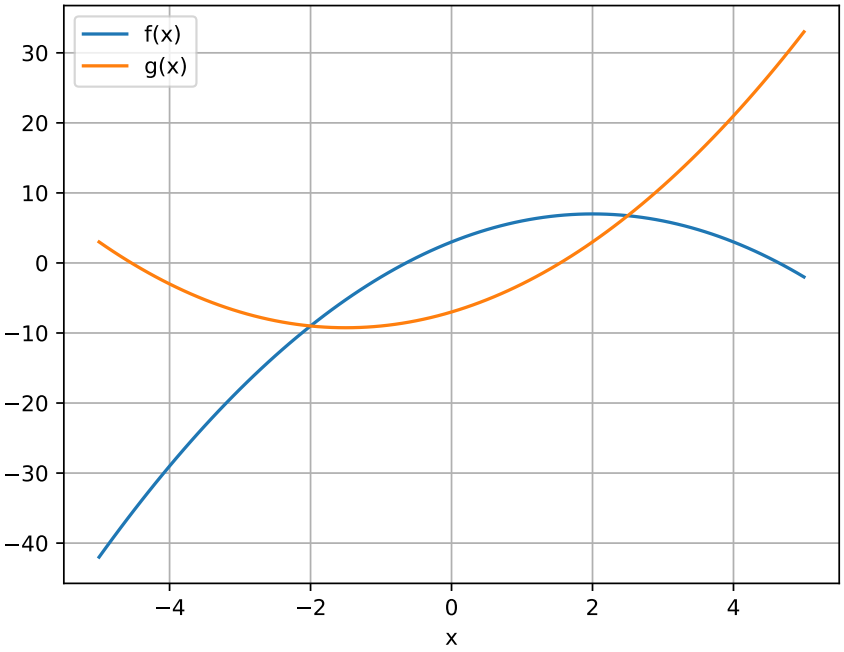
Senere i kapitellet skal vi se hvordan vi kan plotte matematiske funksjoner av denne typen langt mer effektivt, men for nå gjentar vi samme fremgangsmåte som over.

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

```
from pylab import *

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

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

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

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

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

Resultatet blir som vist i Figur 7. Merk at de to kurvene automatisk får ulike farger, og fordi vi brukte `legend()` får vi en nøkkel som knytter de ulike fargene til de ulike kurvene.



Figur 7: To annengradsfunksjoner plottet i samme figur.

5.2 Grafikk i Sypder

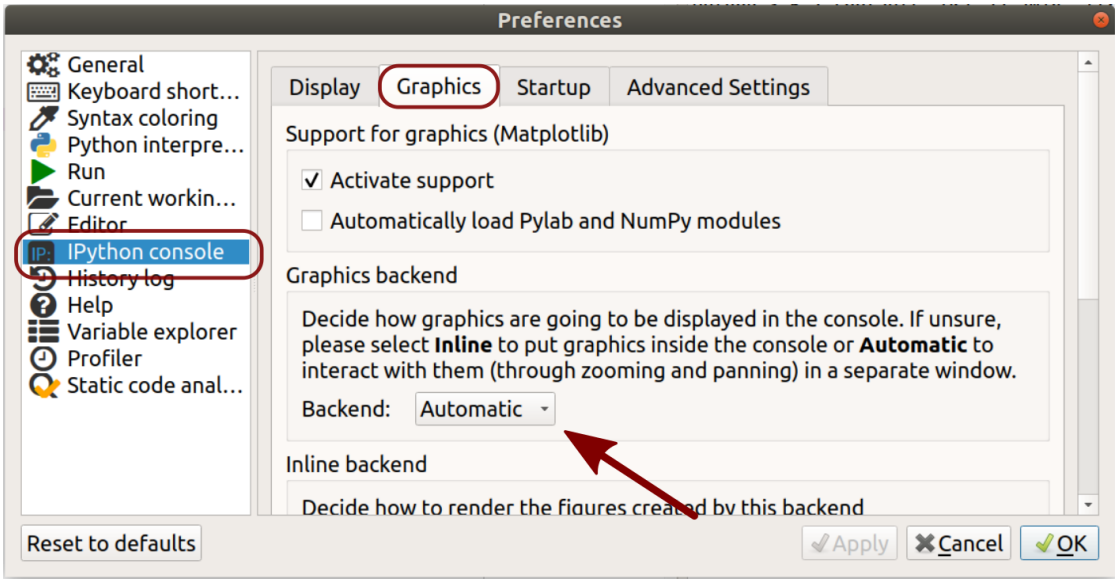
Om du har prøvd det første plotte-eksempelet i Spyder ser du at grafikken dukker opp inne i konsollen, tilsvarende annen output. Dette er på en måte fint, ettersom at det er det man er vandt med. Derimot kan figurene ofte bli for små når de skal passe inn i konsollen, og vi får ikke sett dem skikkelig. I tilegg kan vi ikke interegere med plottene for å f.eks å zoome inn på stilige detaljer og lignende.

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

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

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

Etter at du har restartet Spyder kan du prøve å kjøre programmet ditt på nytt.



Figur 8: Det kan være en god idé å endre innstillinger for hvor figurer vises slik at de ikke havner inne i konsollen. Her kan man også huke av for å automatisk bruke pylab, om man ønsker å slippe å importere i hvert program man bruker.

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

5.3 Lagre plott og grafer

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

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

5.3.1 Lagre bilder ved hjelp av kode

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

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

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

5.3.2 Lagre bilder "manuelt"

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

Om man foretrekker å plotte figurer rett i konsollen kan du høyreklikke på et bilde i konsollen og velge å kopiere eller lagre det. Derimot vil du nok erfare at disse bildene desverre blir litt lav oppløsning. Dette kan du isåfall endre ved å øke oppløsningen under de de samme innstillingene som diskutert over, se under `Inline Backend` i bunn av Figur 8.

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

5.4 Vektoriserte beregninger

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

5.4.1 Vektorer i Python

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

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

Vil du få en feilmelding av typen

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

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

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

```
from pylab import *

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

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

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

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

5.4.2 Bruk av `arange`

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

```
from pylab import *

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

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

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

5.4.3 Bruk av linspace

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

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

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

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

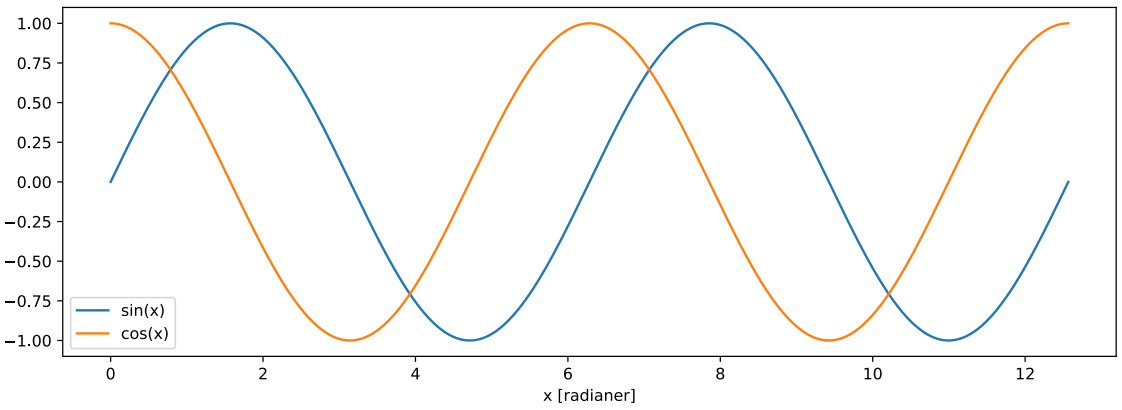
```
x = linspace(-5, 5, 101)
print(x)
```

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

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

```
from pylab import *

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

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

Her plotter vi sinus og cosinus. Ettersom at `pylab` sine trigonometriske funksjoner er definert i form av radianer lar vi x gå fra 0 til 4π. Det koster lite for Python å regne ut mange verdier, så for å få en jevn og fin kurve velger vi like greit 1001 punkter. Resultatet er vist i Figur 9.



Figur 9: Plott av en sinuskurve og en cosinuskurve i samme figur.

5.5 Eksempel: Plott med parameterfremstilling

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

5.5.1 Rett linje

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

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

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

```
from pylab import *

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

# "Pakker ut" koordinatene
x0, y0 = A
x1, y1 = B

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

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

Merk her at vi spesifiserer A og B som verdipar med myke parenteser: `A = (2, 3)`. Dette er en variabeltype som kalles *tupler* i Python, det blir helt ekvivalent med å bruke lister (`A = [2, 3]`), vi bare liker at tuplene ligner litt mer på slik vi skriver

punkter for hånd. Merk at vi pakker ut de to punktene til x- og y-koordinatene for å regne med disse verdiene. Om man syns den delen av koden var lite oversiktlig kunne vi istedet rett og slett definert de fire variablene x_0, y_0, x_1, og y_1 direkte. Resultatet av koden er vist i Figur 10.

5.5.2 Sirkel

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

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

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

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

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

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

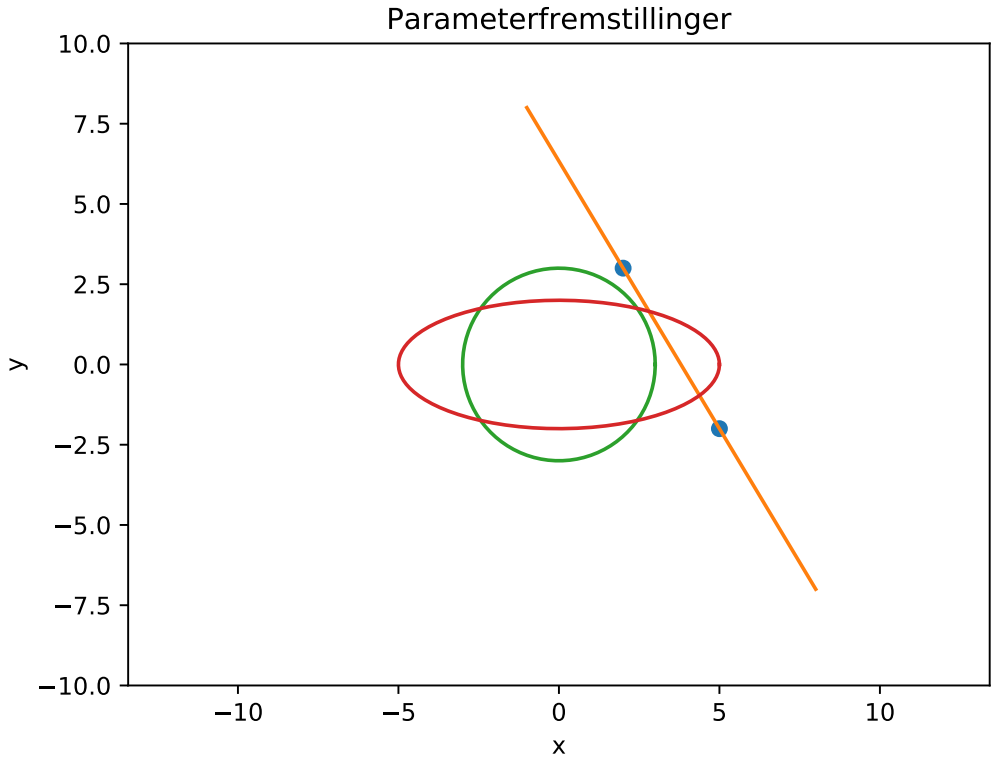
Her har vi lagt til linja `axis('equal')`, denne linja er strengt tatt ikke nødvendig, men om vi dropper den vil sirkelen vår fort se ut som en ellipse fordi x og y-aksen har ulike dimensjoner. Ved å si "axis equal" forteller vi Python at vi ønsker at de to aksene skal være dimensjonert likt, og sirkelen ser ut som en sirkel. Resultatet av koden er vist i Figur 10.

5.5.3 Ellipse

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

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

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

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

Denne er også vist i Figur 10



Figur 10: De tre parameterfremstillingene vist i samme figur.

5.6 Andre former for plotting

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

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

- https://matplotlib.org/gallery/

6 Funksjoner

Vi er nå klare for å dekke det siste nye temaet vi kommer til å dekke i Python-delen av dette kurset, nemlig funksjoner. Funksjoner er et av de mest fundamentale programmeringskonseptene ved siden av variabler, tester og løkker.

Om du har fulgt kompendiet fra starten, så har du faktisk allerede brukt funksjoner hele veien. Hver gang du bruker en kommando av typen `print()`, `sqrt()`, `plot()` og så videre bruker du en funksjon. Felles for disse er at de har ett navn, og at vi bruker parenteser. Noen av disse funksjonene tar argumenter, mens andre gjør ikke det.

En funksjon er, kort fortalt, en bit kode vi kan *kalle* på og bruke gjentatte ganger etter behov. Funksjoner er viktige fordi de forenkler koden vår og gjør den oversiktlig ved å skjule nødvendige detaljer. Ta for eksempel `print()`. Når vi ønsker å skrive ut noe til konsollen vet vi at vi bare trenger å kalle på denne funksjonen. Noen andre har tatt jobben med å programmere og implementere denne funksjonen, slik at vi bare trenger å bruke den.

Man trenger ikke å vite *hvordan* en funksjon fungerer for å bruke den, bare hva den gjør. Dette kalles gjerne for en "black box", med dette mener vi at vi ikke trenger å vite hvordan en maskin ser ut på innsiden for å bruke den, vi trenger bare å vite hva vi skal putte inn av argumenter, og hva vi kommer til å få ut.

6.1 Definere egne funksjoner

I tilegg til å bruke innebygde funksjoner i Python, så kan vi definere egne, og dette er ofte veldig nyttig for å gjøre kode oversiktlig, men også for å generalisere problemer.

La os hoppe rett i det med et eksempel. Si vi skal jobbe med den matematiske funksjonen:

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

Vi har allerede sett hvordan vi kan regne med og plotte slike funksjoner, men la oss nå implementere det som en faktisk Python-funksjon. Det kan vi gjøre som følger:

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

Her bruker vi nøkkelorder `def` (for "define"), og så skriver vi `f(x)` fordi vi velger at funksjonen skal hete `f`, og argumentet skal hete `x`. Vi velger begge disse navnene. Deretter kommer det som vanlig et kolon (`:`) og så et innrykk, tilsvarende det vi har sett for tester og løkker. All koden som har fått innrykk er det som skjer hver gang vi bruker funksjonen ved å kalle på den. I dette tilfellet er innholdet kun én linje.

Til slutt kan du merke at vi har brukt et annet nøkkelord, og dette er `return`. Vi sier at funksjonen *returnerer* en verdi, det er altså den verdien som blir "sendt tilbake" når vi kaller på funksjonen.

Når vi har definert funksjonen skjer det veldig lite i programmet, dette er fordi vi nå har laget en funksjon som er klar for bruk, men vi må faktisk kalle på den før det skjer noe. Vi kan nå bruke funksjonen på lik linje med de innebygde. Vi kan f.eks nå skrive

```
print(f(-2))
print(f(0))
print(f(2))
```

```
-7
-3
9
```

Det som skjer når vi skriver for eksempel `f(0)` er at vi kaller på funksjonen, da blir innholdet i funksjonen utført med argumentet vi har sendt inn `x = 0`. Det er det vi returnerer fra funksjonen som faktisk blir skreve ut.

Alternativt kan vi opprette en varaibel med det funksjonen returner, f.eks:

```
x = 4
y = f(x)
```

Her vil verdien av variabelen `y` bli satt til det funksjonen returner, det vil si, 29.

La oss ta for oss et eksempel til, ballkaster vårt. Vi kan nå definere funksjonen $s(t)$ som

```
s0 = 1.5
v0 = 12
a = -9.81

```

```
def s(t):
    return s0 + v0*t + 0.5*a*t**2
```

Her har vi valgt å definere parametrene `s0`, `v0` og `a` utenfor funksjonen, og det går helt fint. Et alternativt til dette er å gjøre dem til *valgfire* variabler. Isåfall definerer vi funksjonen som

```
def s(t, s0=1.5, v0=12, a=-9.81):
    return s0 + v0*t + 0.5*a*t**2
```

Når vi definerer en funksjon på denne måten sier vi at vi *alltid* må oppgi t når vi kaller på funksjonen, men de andre er det valgfritt å oppgi. Dersom de ikke blir oppgitt så vil standardverdiene vi har skrevet inn bli brukt:

```
print(s(0))
print(s(0, s0=6))
print(s(1, s0=1, v0=9.81))
```

```
1.5
6.0
5.905
```

Nå kan vi for eksempel enkelt plotte to kurver med forskjellige parametre som følger:

```
t = linspace(0, 3.0, 101)

plot(t, s(t), label='"Vanlig" kast')
plot(t, s(t, v0=6), label='"Svakt" kast')

xlabel('Tid [sekunder]')
ylabel('Høyde [meter]')
axis([0, 3, 0, 10])
legend()
show()
```

Man kan også fint definere en funksjon som tar to eller flere input-argumenter. Vi kan for eksempel lage en funksjon som regner ut Pytagoras for oss

```
from pylab import *

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

print(hypotenus(3, 4))
```

```
5.0
```

Det er ikke bare matematiske funksjoner vi kan definere som funksjoner. Vi kan skrive hva som helst av kode inne i en funksjon som vi måtte ønske. La oss for eksempel gå tilbake til eksempelet vi hadde med DNA-strenger. Da oversatte vi en kodestreng av DNA til en malstreng. La oss nå definere en funksjon som oversetter en malstreng av DNA til en mRNA-sekvens, en prosess som gjerne kalles *transkripsjon*.

```
def transkripsjon(malstreng):
    # Lag mRNA fra en DNA malstreng
    mRNA = ""
    for base in malstreng:
        if base == 'A':
            mRNA += 'U'
        elif base == 'T':
            mRNA += 'A'
        elif base == 'C':
            mRNA += 'G'
        elif base == 'G':
            mRNA += 'C'
    return mRNA
```

Vi kan da oversette en gitt DNA-sekvens til mRNA:

```
malstreng = "TAGTGTGCGGGGAACGAGGCTTCTTCTACA"
mRNA = transkripsjon(malstreng)
print(mRNA)
```

```
AUCACACGCCCCUUGCUCCGAAGAAGAUGU
```

Neste steg i prosessen kalles *translasjon*, og går ut på at tre og tre baser i mRNA-strengen oversettes til en gitt aminosyre. Disse aminosyrene tres sammen som perler på en snor og former et protein. Om vi nå definerer en `translasjon()`-funksjon i Python kan vi oversette mRNA-tråden til aminosyrer:

```
malstreng = "TAGTGTGCGGGGAACGAGGCTTCTTCTACA"
mRNA = transkripsjon(malstreng)
polypeptid = translasjon(mRNA)
```

Denne funksjonen tar vi oss ikke tiden til å skrive her og nå, men vi skal se nærmere på dette når vi diskuterer Biologi senere i kurset.

7 Sammensatte eksempler

Nå har vi vært igjennom de viktigste konseptene innen programmering, nemlig variabler, tester, løkker, og funksjoner. I dette kapitellet går vi bare igjennom noen eksempler som kombinerer disse ulike konseptene for å løse litt forskjellige problemstillinger. Disse er ment for å vise måter å kombinere de ulike konseptene for å lage større og mer komplisert oppførsel.

7.1 Skråkast av ball i ulike vinkler

Vi har gjentatte ganger i kompendiet sett på vertikalt kast av en ball. Men la oss nå se på skråkast av ball.

Om man kaster en ball med fart på v_0 i en vinkel α, vil den ha en hastighet med både en vertikal og en horisontal komponent gitt ved

$$\vec{v_0} = (v_0 \cos\alpha, v_0 \sin\alpha).$$

Om vi sier at ballen starter i $x_0 = 0$, men litt over bakken y_0 kan vi gi posisjonen til ballen som parameterfremstillingen

$$\vec{r}(t) = \begin{cases} x(t) = v_0 \cos\alpha \cdot t, \\ y(t) = y_0 + v_0 \sin\alpha \cdot t - \frac{1}{2}gt^2. \end{cases}$$

La oss skrive Python kode som regner ut og plotter bevegelsen til ballen gitt de ulike parametrene. Ettersom at de fleste syns det er lettere å tenke på vinkler i grader enn radianer lar vi vinkelen være gitt i antall grader, og vi konverterer internt til radianer fra formelen

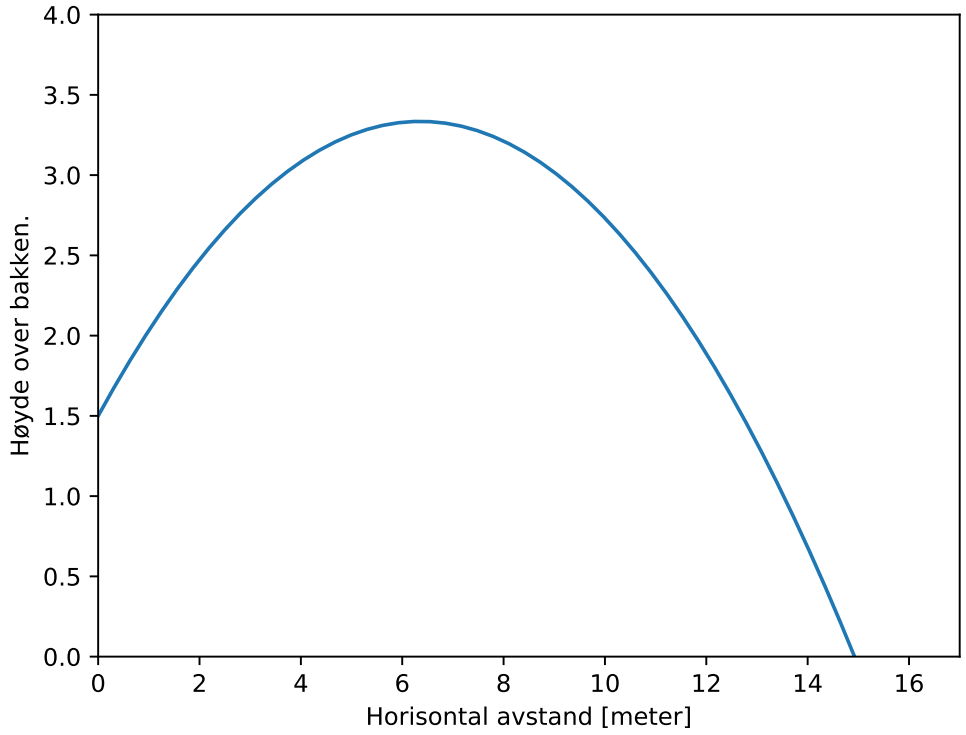
$$\theta = \frac{\alpha}{180}\pi.$$

Koden blir som følger:

```
1 from pylab import *
2
3 alpha = 30
4 v0 = 12
5 y0 = 1.5
6 g = 9.81
```

```
7
8  theta = alpha/180*pi
9
10 t = linspace(0, 3, 101)
11 x = v0*cos(theta)*t
12 y = y0 + v0*sin(theta)*t - 0.5*g*t**2
13
14 plot(x, y)
15 xlabel('Horisontal avstand [meter]')
16 ylabel('Høyde over bakken.')
17 axis([0, 17, 0, 4])
18 show()
```

Her har vi funnet rimelig valg av akser ved prøving og feiling. Resultatet blir som vist i Figur 11.



Figur 11: En ball kastet på skrå i en vinkel av 30°.

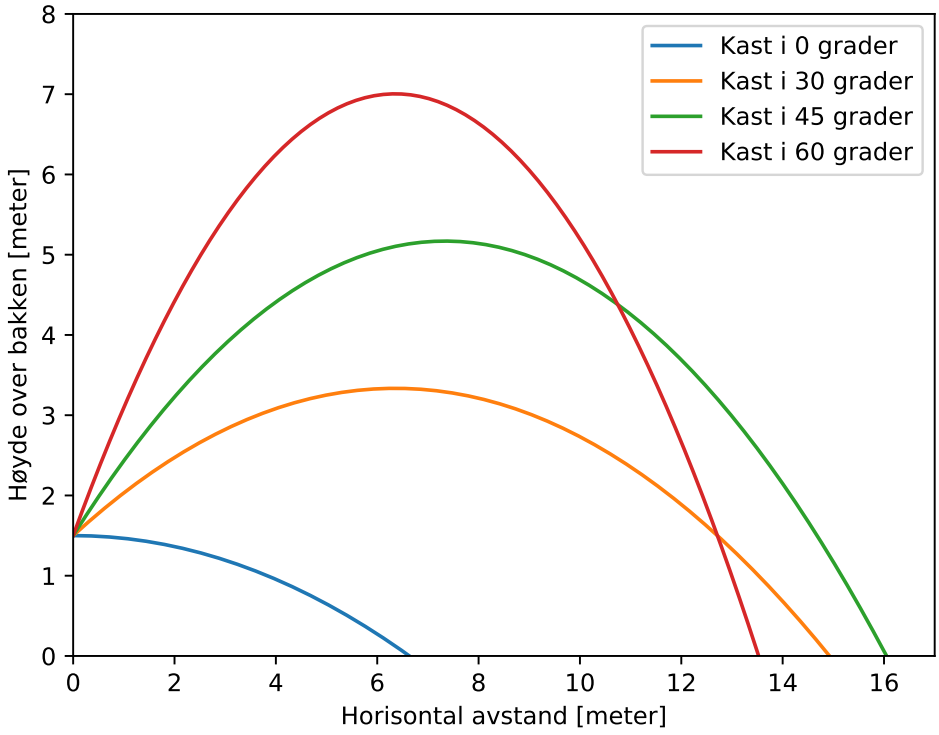
Nå kan vi lure på hvordan kastet endrer seg om man kaster i forskjellig vinkel. Her kan vi for eksempel ønske å plotte ulike kaster i samme figur for å sammenligne enklest mulig. For å gjøre koden oversiktelig lager vi derfor først en funksjon som regner ut selve bevegelsen. Dette er mye av den samme koden som over, bare lagt inn i en funksjon

```
def r(t, alpha=45, v0=12, y0=1.5, g=9.81):
    theta = alpha/180 * pi
    x = v0*cos(theta)*t
    y = y0 + v0*sin(theta)*t - 0.5*g*t**2
    return x, y
```

Når vi har denne funksjonen kan vi enkelt lage og plotte kast i ulike vinkler:

```
t = linspace(0, 3, 101)

for vinkel in 0, 30, 45, 60:
    x, y = r(t, alpha=vinkel)
    plot(x, y, label=f'Kast i {vinkel} grader')

xlabel('Horisontal avstand [meter]')
ylabel('Høyde over bakken.')
axis([0, 17, 0, 8])
legend()
show()
```

Resultatet blir Figur 12.



Figur 12: Ulike skråkast med forskjellig vinkel. Vi observerer at kaster i 45° går lengst, som forventet.

7.2 FizzBuzz

FizzBuzz er en enkel lek man kan leke med to eller flere personer, gjerne litt større grupper. Leken i seg selv er egentlig ment for å lære barn om divisjon og gangetabellen, og har ingenting med programmering å gjøre. Derimot har det blitt til en veldig populær øvelse og kode opp et dataprogram som leker nettopp denne leken. Grunnen til at det er blitt en så populær programmeringsøvelse er at man må beherske både løkker og tester for å få den til, men om man kan programmere er det ikke en så altfor vanskelig øvelse. Derfor har den blitt spesielt populær å bruke i jobbintervjuer, for å sjekke om en kandidat faktisk kan programmere eller ikke.

For å leke FizzBuzz kan man være et par, eller man kan være en 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 man 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*. Målet er å komme helt til 100 uten å gjøre noen feil, og jo raskere man teller jo bedre! Om man gjør feil må man begynne fra start.

Når vi sier at vi skal programmere FizzBuzz 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, men som følger spesielle tileggsregler.

Som alltid når vi programmerer, er det lurt å gå frem i små steg, og sjekke programmet vårt underveis. Det første vi gjør er derfor å lage et program som teller opp til 100:

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

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

Nå legger vi til den første regelen, nemlig at hvert tall delelig med 3 skal erstattes med *Fizz*. Dette får vi til med en if-test. For å sjekke om et tall er delelig bruker vi modulo-operatoren.

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

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

Nå går vi et steg til, og legger også inn testen med om tallet er delelig på 5. Siden vi nå skal ha tre mulige utfall i testen vår må vi bruke `elif`, som stod for "else if":

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

    elif tall % 5 == 0:
        print('Buzz')

    else:
        print(tall)
```

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

Fra utskriften ser vi at det ser ut til å fungere bra for både 3-gangen og 5-gangen. Det er nå lett og tro at vi er ferdig, men det er en liten justering vi trenger. Husk at dersom et tall er delelig med *både* 3 og 5, så skal det skrives ut begge deler, altså "FizzBuzz". programmet vi har skrevet vil kun skrive ut "Fizz". Dette er slik en if-elif-else fungerer i Python, kun én av tilfellene vil treffe inn.

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

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

    # Eller er det delelig med bare 3?
    elif tall % 3 == 0:
        print('Fizz')

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

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

En annen løsning ville her vært å isteden først sjekke om tallet var delelig på 15, fordi fordi $3 \times 5 = 15$.

Da er vi egentlig ferdig med oppgaven. FizzBuzz er blitt en populær programmeringsoppgave fordi selve oppgaven er så enkel å forstå, men for å programmere det må man beherske både løkker og tester. Derimot er det også en fin oppgave fordi man programmere den opp stegvis, og ved hvert steg sjekke om vi er på rett vei eller ikke.

Her kan vi også nevne at man kan gå frem på andre måter for å løse oppgaven. Se for eksempel på denne koden og prøv å skjønn hva som skjer her

```
for tall in range(1, 101):
    utskrift = ""

```

```
    if tall % 3 == 0:
        utskrift += "Fizz"
    if tall % 5 == 0:
        utskrift += "Buzz"

    if utskrift == "":
        utskrift = tall

    print(utskrift)
```

Denne koden går frem på en litt annen måte for å løse det samme problemet. Hvilken av løsningene som er "best" finnes det ikke noe fasitsvar på.

7.3 Finne Primtall

En annen god oppgave å programmere er et program som kan finne primtall. Denne øvelsen krever en del programmeringsforståelse og er innom de fleste av konseptene vi har dekket i dette kompendiet.

Vi minner om at et primtall, per definisjon, er et positivt heltall som kun er delelig med 1 og seg selv. Merk også at 1 selv ikke regnes som et primtall.

Om du nå får et tall, si, 121. Hvordan sjekker du om dette er et primtall? Utifra definisjonen må vi sjekke om 121 er delelig med noen av tallene i rekka $2, 3, 4, \ldots 120$. Om vi finner ett eneste tall som deler 121, så er det ikke et primtall. Om vi jobber oss gjennom hele rekka uten å finne en slik divisor, så er 121 et primtall.

Vi skal nå programmere en funksjon vi kan bruke til å sjekke et hvilket som helst tall. Vi kaller denne funksjonen `er_primtall(n)`:

```
def er_primtall(n):
```

For å sjekke tallet n må vi løkke igjennom tallrekka $[2, n)$ og sjekke om noen av tallene deler n.

```
for d in range(2, n):
    if n % d == 0:
        ...
```

Hvis denne testen slår inn, så har vi funnet et tall som deler n, og vi er egentlig ferdige, fordi vi vet at vi ikke har et primtall. Derfor velger vi å returnere verdien `False`. Dette er et spesielt nøkkelord i Python.

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

Merk spesielt her at så fort programmet kommer til en `return` kodelinje, så er funksjonskallet ferdig, og ikke noe mer vil skje inne i funksjonen. Dette er bra, fordi om vi har funnet ut at n ikke er et primtall er det ingen grunn til å fortsette med hele resten av løkka, det ville bare være lite effektivt.

Men hva om vi kommer oss igjennom hele løkka uten å finne noen tall som deler n? Jo, da er jo n et primtall, og vi kan returnere `True` istedet.

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

La oss nå prøve denne funksjonen med et par utvalgte verdier:

```
for tall in range(1, 6):
    print(tall, er_primtall(tall))
```

```
1 True
2 True
3 True
4 False
5 True
```

Vi ser at funksjonen vår fungerer bra, med ett unntak, nemling verdien $n = 1$. Denne sier funksjonen vår at er et primtall. Prøv å skjønn hvorfor.

Måten vi løser dette problemet er rett og slett ved å håndtere $n = 1$ eksplisitt, siden akkurat denne verdien er litt spesiell

```
def er_primtall(n):
    if n == 1:
        return False
```

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

Med dette er funksjonen vår komplett, og den kan i prinsippet sjekke et hvilket som helst positiv heltall n, for eksempel 121:

```
if er_primtall(121):
    print("121 er primtall")
else:
    print("121 er ikke et primtall")
```

```
121 er ikke et primtall.
```

Vi kan også sjekke langt større tall, f.eks 40193, som tilfeldigvis er et primtall. Om du prøver veldig store tall vil ikke løsningen vår være effektiv nok, og datamaskinen vil måtte regne veldig lenge for å komme frem til et svar. Det finnes derimot langt mer effektive måter å sjekke om tall er primtall på, som vi kan bruke om vi er interessert i så store primtall.

7.4 Gjettespill

Over/under er en enkel gjettelek to personer kan leke, som også egner seg godt til å programmere. To personer spiller. Person A tenker på et tilfeldig tall mellom 1 og 1000, deretter skal person B prøve å gjette seg frem til tallet. Etter hvert forslag B gir, så skal person A si om gjetter var over, under, eller helt riktig. På denne måten får B mer og mer informasjon for hvert gjett, og kan peile seg inn til riktig svar.

La oss nå prøve å kode opp dette spillet. Først må vi definere hvilket tall datamaskinen tenker på. Her skal vi etterhvert vise hvordan datamaskinen kan velge et tall tilfeldig, ellers er ikke spillet noe gøy. Mens vi prøver å programmere det derimot, kan vi jukse litt å velge en bestemt verdi, slik at det er lettere å teste programmet vårt.

```
fasitsvar = 456
print("Jeg har tenkt på et tall mellom 1 og 1000.")
print("Prøv å gjett det!")
```

```
print()
gjett = int(input("Gjett: "))
```

For å la brukeren gjette bruker vi `input()`, og vi må konvertere svaret brukeren gir til en tallvariabel med `int()`.

Vi skal nå gjenta den samme prosessen mange ganger: nemlig la brukeren gjette. For å gjenta noe mange ganger bruker vi en *løkke*. Hvor lenge skal vi fortsette, jo, helt til brukeren gjetter riktig. Vi vet ikke på forhånd hvor mange gjett dette vil ta, derfor lønner det seg å bruke en while-løkke. Inne i løkka må vi bruke en if-test for å gi informasjon til brukeren om gjetter er for høyt eller for lavt

```
while gjett != fasitsvar:
    if gjett > fasitsvar:
        print(f"Ditt gjett på {gjett} er for høyt.")
    else:
        print(f"Ditt gjett på {gjett} er for lavt.")
    gjett = int(input("Gjett: "))
```

Merk at vi til slutt i løkka må la brukeren gjette på nytt, ellers vil vi får en uendelig løkke (hvorfor det?).

Løkka vil gjenta seg selv helt frem til betingelsen blir falsk, det vil si helt frem til brukeren gjetter riktig tall. Da kan vi skrive ut en hyggelig melding.

Vi kan nå skrive programmet vi har så langt og teste det

```
fasitsvar = 456
print("Jeg har tenkt på et tall mellom 1 og 1000.")
print("Prøv å gjett det!")
print()
gjett = int(input("Gjett: "))

while gjett != fasitsvar:
    if gjett > fasitsvar:
        print(f"Ditt gjett på {gjett} er for høyt.")
    else:
        print(f"Ditt gjett på {gjett} er for lavt.")
    gjett = int(input("Gjett: "))

print(f"Det stemmer! {gjett} er helt riktig!")
```

Vi prøver nå programmet vårt. Det hjelper å vite svaret mens vi holder på

```
Jeg har tenkt på et tall mellom 1 og 1000.
Prøv å gjett det!

Gjett: 400
Ditt gjett på 400 er for lavt.
Gjett: 500
Ditt gjett på 500 er for høyt.
Gjett: 450
Ditt gjett på 450 er for lavt.
Gjett: 456
Det stemmer! 456 er helt riktig!
```

Nå gjennstår det egentlig bare å bytte ut fasitsvaret med en kodelinje så det blir tilfeldig hver gang. For å gjøre dette kan vi bruke funksjonen `randint(a, b)` fra pylab, denne trekker et tilfeldig heltall fra intervallet $[a, b)$ ("randint" står for "random integer").

I tilegg ønsker vi gjerne å holde styr på antall gjett brukeren har gitt. Derfor legger vi også til en tellevariabel. Programmet blir nå som følger:

```
1  from pylab import *
2
3  fasitsvar = randint(1, 1001)
4  print("Jeg har tenkt på et tall mellom 1 og 1000.")
5  print("Prøv å gjett det!")
6  print()
7  gjett = int(input("Gjett: "))
8  antall_gjett = 1
9
10 while gjett != fasitsvar:
11     if gjett > fasitsvar:
12         print(f"Ditt gjett på {gjett} er for høyt.")
13     else:
14         print(f"Ditt gjett på {gjett} er for lavt.")
15     gjett = int(input("Gjett: "))
16     antall_gjett += 1
17
18 print(f"Det stemmer! {gjett} er helt riktig!")
19 print(f"Du brukte {antall_gjett} totalt!")
```

7.4.1 Strategi

Bare det å kode opp over/under er en god øvelse for å lære programmering og algoritmisk tankegang, det kan også være morsomt. I mattetimen kan vi gå et steg lenger, og nå tenke på hvordan vi bør *spille* dette spillet for å gjøre det best mulig. Altså, hvilken strategi skal vi velge?

Her kan vi for eksempel velge å bruke *midtpunktsmetoden*, der vi alltid velger midtpunktet av det intervallet vi har igjen. Da starter vi altså med å gjette 500. Om vi er for lave går vi opp til 750, om vi er for høye går vi ned til 250.

Midtpunktsmetoden er en god strategi, fordi uansett om vi er for høye eller lave kan vi eliminere halve det intervallet som er igjen. Et annet godt navn for denne strategien er *halveringsmetoden*.

Her kan det for eksempel være en god øvelse å regne seg frem til hvor mange gjett man trenger for å *garantert* komme frem til riktig svar med midtpunktsmetoden, som blir

$$\log_2(1000) \simeq 10.$$

Her kan det nevnes at midtpunktsmetoden ikke bare er en god strategi for over-/under leken, men en god matematisk metode for å løse matematiske ligninger. Dette skal vi se nærmere på i neste del av kurset.

8 Oppgaver

Programmering læres ved å gjøre. Dette kapitellet består av oppgaver du kan løse for å øve på programmeringskonseptene gitt i dette kompendiet. Oppgavene er også lagt på et slikt nivå at de kan brukes i egen undervisning etter interesse. Vi anbefaler å jobbe dere igjennom oppgavene i par, på den måten kan dere diskutere og hjelpe hverandre.

Merk at noen av oppgavene er knyttet opp mot bestemte programfag. De varierer også i omfang og vansklighetsgrad. Det er derfor fritt frem for å hoppe over oppgaver utifra egen interesse.

Om du ønsker flere oppgaver er det bare å ta kontakt med oss, så kan vi supplere med eksterne kilder. Vi kan også nevne nettsiden Project Euler. Dette er en nettside som gir oppgaver fra matematikk som er spesielt tiltenkt å løses med programmering. Disse starter forholdsvis enkelt, men stiger i vansklighetsgrad etterhvert. Du finner oppgavene her:

- `https://projecteuler.net/archives`

8.1 Variabler og Utregninger

Oppgave 1 (Printing)

a) Lag et program som skriver ut teksten "Hei, Verden!", til skjermen.

b) Endre programmet slik at du først lagrer navnet ditt i en variabel, og så får programmet ditt til å skrive ut en hilsen direkte til deg.

c) Endre programmet slik at du først bruker `input()`-funksjonen til å først spørre brukeren om navnet dems, og deretter skriver ut en beskjed som bruker navnet de har oppgitt.

Oppgave 2 (Volumberegninger)
Formelen for volumet til en kule er gitt ved

$$V = \frac{4}{3}\pi R^3.$$

a) Jorda er tilnærmet en perfekt kule, og har en radius på omtrent 6371 km. Regn ut volumet av jorda.

b) Gjenta beregningen din, men denne gangen endre enheter slik at du får svaret oppgitt i antall liter.

c) Jorda har en total masse på omtrent

$$M = 5.972 \times 10^{24} \text{ kg}.$$

Regn ut den gjennomsnittlige massetettheten til jorda i kg per liter. Virker svaret ditt rimelig?

Oppgave 3 (Regne ut pH)
pH-verdien til en væske eller løsning er et mål på hvor sur væsken er, det vil si konsentrasjonen av hydrogenioner $[H^+]$. Formlene for å regne med pH er gitt ved

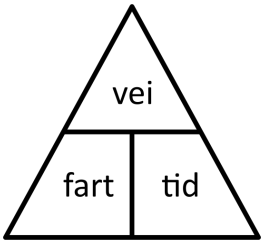
$$\text{pH} = -\log_{10}([H^+]), \qquad [H^+] = 10^{-\text{pH}}.$$

a) Regn ut konsentrasjonen av hydrogenioner for en væske med pH 3.6.

b) Regn ut pH verdien til en væske med konsentrasjon av hydrogenioner lik $4.7 \cdot 10^{-5}$ mol/L.

c) Cola har en pH på ca 2.5, nøytralt vann har en pH på 7. Hvor mange ganger fler hydrogenioner er det i cola sammenlignet med vann?

Oppgave 4 (Vei-Fart-Tid)
Vei-fart-tid formelen er grunnlaget for bevegelsesligningene i Fysikken. Egentlig er det tre formler i et, for vi kan stokke om på den avhengig av hva vi ønsker å regne ut. Av og til tegnes vei-fart-tid formelen som en pyramide:



a) Skriv opp uttrykket for å regne ut vei, gitt fart og tid. La fart være gitt i km/h og tid i minutter og avstand i kilometer.

b) Gjenta prosessen for de to andre formene av vei-fart-tid formelen.

Oppgave 5 (Konsentrasjonskalkulator)
Når man gjør eksperimenter i kjemien må man ofte lage løsninger med en bestemt molar konsentrasjon.

Si at vi ønsker å lage en løsning av et stoff X, som veier M g/mol. Vi ønsker å lage en løsning av volum V mL av en gitt konsentrasjon c mol/L.

a) Lag et program som regner ut hvor mange gram av stoff X man må veie ut og løse opp, oppgitt med en nøyaktighet på 0.01 gram.

Hint: For å skrive ut en variabel med 3 desimaler bruk print-formatering med `'{m:.2f}'`.

b) Test programmet ved å finne mengden bordsalt (NaCl) du trenger for å lage 100 mL løsning med 2.5 mol/L.

c) Ved 25°C klarer man å løse opptil 35.7 gram NaCl per 100 mL vann, etter dette er løsningen metter. Øk konsentrasjonen c i programmet ditt til du er omtrent på denne grensa, hva slags konsentrasjon svarer dette til?

Oppgave 6 (Hardy-Weinbergs Lov)
Anta at et gitt gen finnes i to utgaver (kalt alleler), der den ene er den ene er dominant (A), og den andre recessiv (a). Mennesker er diploide organismer, som betyr at vi har to sett kromosomer. Et gitt menneske har derfor én av kombinasjonene AA, Aa eller aa, disse kaller vi de ulike genotypene.

Hardy-Weinbergs lov lar oss regne ut frekvensen av de ulike genotypene, gitt frekvensen av de to allelene, under idealiserte betingelser. Om frekvensen av A er gitt ved p og frekvensen av a er gitt ved q (merk at $p + q = 1$) har vi at

- Frekvensen av genotype AA: p^2.
- Frekvensen av genotype Aa: $2pq$.
- Frekvensen av genotype aa: q^2.

a) Lag et program som gitt en p og en q først sjekker at $p+q = 1$. Dersom dette er tilfellet skal det regne ut frekvensen av de ulike genotypene og skrive dem ut. Dersom $p + q \neq 1$ skal programmet skrive ut en feilmelding.

Sigdcelleanemi er en genetisk sykdom forårsaket av en variant i genet for hemoglobin. Vi kaller det vanlige genet for S og sigdcellegenet for s. Sykdommen er recessiv, man må altså være homozygot for sigdcelle (ss) for å få alvorlig sykdom. De heterozygote (Ss) er bærere av genet, men utvikler ikke sigdcellesykdom. Derimot er det slik at alle med sigdcellegenet, både (Ss) og (ss) genotypen, er resistente mot malaria.

Sigdcelle er en alvorlig sykdom, og man ønsker ikke (ss) genotypen. Derimot er det altså i visse tilfeller (f.eks ved malariaforekomst) hvor det er en fordel å være bærer av genet (Ss) fremfor å ikke ha det i det heletatt (SS).

b) Anta først at allelefrekvensen av sigdcellegenet er $q = 0.1$ og kjør programmet ditt. Øk deretter frekvensen til $q = 0.3$. Sammenlign genotypefrekvensene du får ut. Kan du forklare hvorfor man observerer at allelefrekvensen av sigdcelle typisk er signifikant høyere i områder med malaria enn andre deler av verden?

8.2 If-tester

Oppgave 7 (abc-formelen)
En generell annengradsligning av typen

$$ax^2 + bx + c = 0,$$

vil kunne ha ingen, én eller to, løsninger avhengig av verdiene til a, b og c. Dette bestemmes av *diskriminanten*:

$$d = b^2 - 4ac,$$

hvis $d > 0$ har vi to løsninger, hvis $d = 0$ har vi én løsning, og om $d < 0$ har vi ingen løsninger.

a) Bruk en if-test til å sjekke hvor mange løsninger det finnes for en gitt a, b, og c.

Løsningene finnes ved selv abc-formelen, som er gitt ved

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

b) Utvid programmet ditt så det også skriver ut løsningene.

c) Test programmet ditt ved å sjekke følgende tilfeller

- $a = 1$, $b = 2$, $c = 1$
- $a = 2$, $b = 3$, $c = -9$
- $a = -\frac{1}{2}$, $b = 0$, $c = 2$

d) Modifiser programmet ditt så det spør brukeren om verdier for a, b og c og skriver ut de tilhørende løsningene.

Oppgave 8 (Bohr's Atommodell)
Bohr's atommodell kan forutsi energinivåene til de ulike elektronskallene. Når et elektron deeksiteres fra skall n til m (der $m < n$) vil det sleppe ut et foton med energien:

$$\Delta E = E_0 \left(\frac{1}{m^2} - \frac{1}{n^2} \right) k,$$

der $E_0 = 13.6$ eV.

a) Skriv et program som finner energien til et foton for en vilkårlig m og n.

Man kan finne bølgelengden til et foton med energien E fra formelen

$$\lambda = \frac{hc}{E},$$

der $hc = 1240$ eV.

b) Inkluder denne konversjonen i programmet ditt, slik at den for en gitt m og n skriver ut både energien til fotonet, og bølgelengden.

c) Bruk en if-test til å også skrive ut fargen på lyset til fotonet som blir sendt ut. Her kan du skrive av koden til det tilsvarende eksempelet vist i seksjon 3.2.

d) Bølgelengdene vi finner om vi velger $m = 2$, og ulike n kalles for Balmerserien. Finn de bølgelengdene vi finner i Balmerserien som er synlig lys.

e) Spektrallinjene i Balmerserien er vist i dette bildet:

Passer det du ser med svarene du fikk i oppgave (d)?

8.3 Løkker

Oppgave 9 (Trekanttall)
Et trekanttall er summen av tallrekken

$$T_n = 1, 2, \ldots, n.$$

For eksempel det femte trekanttallet:

$$T_5 = 1 + 2 + 3 + 4 + 5 = 15.$$

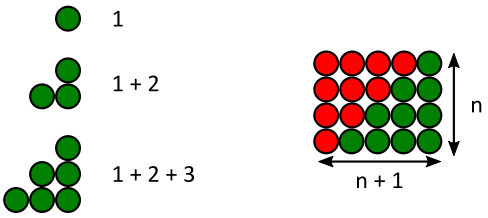
a) Lag et program som regner ut T_{1000} ved hjelp av en for-løkke.

En populær historie skal ha det til at matematikeren Carl Friedrich Gauss som barn fikk denne oppgaven av sin mattelærer når han gikk på skolen på 1700-tallet. Den slemme læreren ville bare holde Gauss opptatt en stund med å legge sammen en haug tall. Vår lure Gauss derimot, utledet istedet formelen

$$T_n = \frac{n(n+1)}{2}.$$

b) Bruk formelen til Gauss og sjekk at du får samme svar som programmet ditt for T_{1000}.

For å skjønne hvorfor denne formelen fungerer kan det lønne seg å tenke over hvorfor de kalles *trekanttall*. Om vi tegner opp summene som antall baller, og tenger først 1, så 2, og så 3 bortover, sånn som dette:



Så ser vi at T_n former en trekant. Vi ser også at denne trekanten vil være halvparten av en firkant med sider n og $(n+1)$. Dermed vil arealet av trekanten, som er trekanttallet, være gitt ved formelen til Gauss.

Oppgave 10 (Riskorn og Sjakkbrett)
Om vi plasserer ett riskorn på den første ruta på et sjakkbrett, så to riskorn på neste rute, så fire på den tredje ruta og så videre: Hvor mange riskorn får vi til sammen? Dette kan skrives som summen

$$1 + 2 + 4 + 8 + ...$$

a) Bruk en for-løkke og `range`-funksjonen til å løkke over alle sjakkbrettets ruter og finn antall riskorn vi får totalt.

b) Bruk print-formattering til å skrive svaret ved hjelp av vitenskapelig notasjon (for eksempel: `"{total:.1e}"`).

c) Det er omtrent 50000 riskorn i en 1 kg sekk. Bruk en while-løkke til å finne antall ruter vi må inkludere for å få minst 1 kg ris.

Oppgave 11 (Fibonaccirekka)
Fibonaccirekka starter

$$1, 1, 2, 3, 5, 8, 13, \ldots$$

der hvert nye tall i rekka finnes ved at de to forrige summeres.

a) Definer to variabler, `a=0` og `b=1`.

b) Lag en while-løkke med betingelsen $b < 100$.

c) Inne i løkka, regn først ut det neste tallet i Fibonaccirekka: `ny = a + b`. Skriv så dette nye tallet ut og oppdater så a og b så de har verdiene til de to forrige tallene i rekka.

d) Endre betingelsen i rekka så du skriver ut alle Fibonacci-tallene som er mindre enn 1000.

e) Forholdet mellom to tall som kommer etterhverandre i Fibonaccirekka går nærmere og nærmere det gylne snitt. Regn ut forholdet mellom de to siste tallene du fant i forrige oppgave for å estimere det gyldne snitt.

Oppgave 12 (Skuddår)
Du skal nå skrive et program der brukeren gir et årstall, og programmet forteller oss om det er et skuddår eller ikke. Derimot er reglene for hvilke år som er skuddår litt kompliserte, så vi tar det steg for steg.

a) I første omgang sier vi at alle årstall som er delelige med 4 er skuddår, og alle de andre er ikke det. Lag en if-test som sjekker dette. Sjekk koden med at 2016 var et skuddår, mens 2018 ikke var det.

b) Nå legger vi til at alle årstall som er delelig med 100 er et unntak til den første regelen, disse årene er altså *ikke* skuddår. Legg til dette i koden ved hjelp av en ny `if` eller `elif`-test. Sjekk koden din ved at år 1700, 1800 og 1900 alle ikke er skuddår.

c) Nå legger vi til et unntak til unntaket, om at alle årstall som er delelige med 400 *er* skuddår. Legg også til dette i koden din. Sjekk nå at 1700, 1800 og 1900 fortsatt ikke er skuddår, men at år 2000 er det.

d) Om det bare var slik at hvert fjerde år var et skuddår hadde et gjennomsnittlig år vært 365.25 dager. Men hvordan blir det med tilleggsreglene? Bruk en løkke til å legg sammen alle år fra og med år 0 til og med år 1999. Hvor mange dager har vært år i gjennomsnitt?

e) Skuddår finnes så antall gjennomsnittsdager per år skal ligge så tett opptil det astronomiske året. Jorda bruker 365.2422 døgn på å gå rundt sola én gang. Hvor nært dette kommer vi med tilleggsreglene?

Oppgave 13 (Gangetabellen)
Den følgende koden skriver ut gangetabellen opp til og med 10×10.

```
for i in range(1, 11):
    for j in range(1, 11):
        print(f"{i*j:5}", end="")
    print()
```

a) Kopier koden over til Spyder og kjør den.

b) Gå igjennom koden steg for steg å prøv å forstå hvordan den virker. Gjør gjerne dette i par.

8.4 Plotting

Oppgave 14 (Dempet svingning)
Plott funksjonen

$$4\cos(4x)e^{-x/2\pi},$$

for $x \in [0, 8\pi]$.

Oppgave 15 (Parameterfremstilling av Sirkel)
I seksjon 5.5 lagde vi en parameterfremstilling av en sirkel. Men vi lot sirkelens sentrum være origo. Endre på programmet så du kan kontrollere sirkelens sentrum (x, y).

Oppgave 16 (Grafisk løsning av ligninger)
I denne oppgaven skal vi løse ligningen

$$f(x) = g(x),$$

grafisk ved hjelp av plotting. Der $f(x)$ og $g(x)$ er to annengradsligninger:

$$f(x) = x^2 - 2x - 4,$$
$$g(x) = -2x^2 - 4x + 8.$$

a) Bruk `linspace` til å definere en rekke verdier $x \in [-5, 5]$.

b) Regn ut kurvene `f(x)` og `g(x)`.

c) Plott de to kurvene i samme figur

d) Finn alle x-verdier der de to kurvene skjærer hverandre. Dette kan du gjøre ved å holde musepekeren over skjæringspunktet i figurvinduet og lese av koordinaten.

I dette tilfellet er det lett å løse $f(x) = g(x)$ for hånd også, eller ved å bruke abc-kalkulatoren vi lagde i en tidligere oppgave. Derimot kan grafisk løsning av ligninger også brukes for ligninger det ikke finnes noen analytisk løsning for.

8.5 Funksjoner

Oppgave 17 (Absoluttverdi)
Et reelt tall består av et fortegn og en tallverdi, kalt *absoluttverdi*. Når vi finner absoluttverdien til et tall "fjerner vi fortegnet". Det betyr at absoluttverdien til et tall alltid er positiv. Absoluttverdien til et tall a skrives $|a|$ og er definert som:

$$|x| = \begin{cases} x, & \text{hvis } x \geq 0 \\ -x, & \text{hvis } < 0 \end{cases}$$

a) Definer en funksjon `absoluttverdi(x)` som bruker en if-test til å finne og returnere absoluttverdien til x.

b) Lag en funksjon `størst_abs(a, b)` som tar de to tallene a og b og returnerer det tallet som har størt absoluttverdi.

Oppgave 18 (Konvertering av temperatur)
I Norge oppgir vi temperaturer i målestokken Celsius, men i USA bruker de som oftest Fahrenheit. De fleste oppskrifter for eksempel, sier at kaken skal bakes ved 350°F, men hvor mye er egentlig det?

For å regne over fra Fahrenheit til Celsius bruker vi formelen:

$$C = \frac{5}{9}(F - 32).$$

Der F er antall grader i Fahrenheit, og C blir antall grader i celsius.

a) Lag en funksjon `F2C`, som tar en temperatur i antall grader Fahrenheit inn, og returnerer den tilsvarende temperaturen i grader Celsius.

b) Bruk programmet ditt til å finne ut hvor mange grader Celsius 350°F svarer til. Virker det rimelig å skulle bake en kake ved denne temperaturen?

c) Snu på ligningen for hånd, så vi løser for F istedenfor C. Lag en funksjon `C2F` som gjør den motsatte konverteringen.

d) Bruk funksjonen til å finne frysepunktet og kokepunktet til vann i Fahrenheit målestokken.

Oppgave 19 (Kombinatorikk)
Vi skal nå definere tre funksjoner som er nyttige i kombinatorikk.

Vi ser først på fakultet, som er antall måter å ordne n ting på, altså antall permutasjoner. Fakultet skrives $n!$, og er gitt ved produktet av alle tall fra og med 1 opp til og med n.

a) Lag en funksjon `fakultet(n)`, som tar et heltall n inn, og returnerer $n!$.

b) Det er 5 elever som er blitt trukket ut til muntlig eksamen. Hvor mange forskjellige rekkefølger kan de eksamineres i?

Den neste funksjonen vi skal se på er *kombinasjoner uten tilbakelegging*, som skrives $n\mathrm{C}k$. Dette er antall måter vi kan velge ut k elementer utifra n, dersom rekkefølgen ikke er viktig. Denne funksjonen er gitt ved binomialkoeffesienten:

$$n\mathrm{C}k = \binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

c) Definer denne funksjonen ved å benytte deg av fakultetsfunksjonen du nettopp lagde.

d) Vi skal plukke 3 elever fra en klasse på 28 til en skriftlig eksamen. Hvor mange ulike kombinasjoner av elever kan vi ende opp med å trekke?

Til slutt skal vi lage funksjonen for antall *permutasjoner uten tilbakelegging*. Dette skrives $n\mathrm{P}k$ og er antall måter å velge ut k elementer fra n, dersom orden er viktig. Formelen er gitt ved

$$n\mathrm{P}k = \frac{n!}{(n-k)!}.$$

e) Definer denne funksjonen ved igjen å bruke fakultetsfunksjonen din.

f) Utifra den samme klassen på 28 elever, hvor mange måter er det å velge en hovedrepresentant, en første vara, og en andre vara til elevrådet?