Programmering i skolen:
Et kræsjkurs i Python for realfagslærere

Del 1: En kort innføring i Python

kodeskolen simula

kodeskolen@simula.no

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

I løpet av kurset vil vi gi en kort innføring i hvordan man kan bruke det tekstbaserte programmeringspråket Python som et verktøy i eget klasserom. Grunnet begrenset med tid vil vi ikke prøve å gi en heldekkende innføring i programmering, eller generelle bruksområder. Vi prøver isteden å rette oss inn mot tema som dekkes i matematikken på VGS.

Kurset, og dokumentet, er delt i to hoveddeler. Det første dekker en kjapp innføring til selve programmering. Her vil dere se hvordan man kan skrive og kjøre kode, og et par grunnlegende programmeringsprinsipper. I den andre delen går vi inn på et konkret opplegg for matematikk med numerisk løsning av likninger.

Innhold

1 Vi setter igang

1.1 Ditt første program

I programmeringsverden er det vanlig å lage et såkalt "Hello, World!" som sitt første program. Et slikt program er ikke fryktelig komplisert, det skal bare skrive ut en beskjed når det kjøres. De som er helt ferske til programmering skriver ofte dette programmet, men det gjør også erfarne programmerere, for eksempel hvis de skal lære seg et nytt programmeringsspråk. "Hello, World!" programmet er som kodeverdens svar på å si "Hei, mitt navn er..." på et nytt språk. I Python er dette programmet ganske enkelt, det er faktisk kun en enkelt kodelinje:

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

Her bruker vi funksjonen `print` til å skrive ut en beskjed. Med `print` mener vi ikke å printe eller skrive ut til papir, men til skjermen. Denne funksjonen er altså det vi bruker for å få et program til å skrive beskjeder til oss når vi kjører programmene.

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.

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.

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.

1.2 Printing

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

Et program kan godt inneholde flere `print` kommandoer på rad, da vil hver beskjed

havne under hverandre. Vi kan også legge inn mellomrom i beskjeden for å påvirke hvor ting havner.

```
print('Hei')
print('   på')
print('      deg!')
```

```
Hei
   på
      deg!
```

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

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

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

1.2.1 Å printe enkel grafikk med ascii-kunst

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

Her skriver vi ut en liten elefant

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

Vår erfaring er at de fleste elever syns det er morsomt å lage enkle programmer som gjør små oppgaver og kommuniserer ved hjelp av *print*. Noen elever er derimot veldig opptatt av grafikk. Her kan Ascii-kunst brukes til å være litt kreativ. Å lage ascii-kunst fra bunn av kan være tidkrevende, men det finnes drøssevis man kan finne på nett om man søker på "ascii art".

Her har vi for eksempel et stemningsfullt bilde vi fant med søket "ascii art cat":

```
                    *      ,MMM8&&&.              *
                           MMMM88&&&&&     .
                          MMMM88&&&&&&&&
                *         MMM88&&&&&&&&
                          MMM88&&&&&&&&
                          'MMM88&&&&&&'
                            'MMM8&&&'       *
                  |\___/|
                  )     (             .              '
                 =\     /=
                   )===(       *
                  /     \
                  |     |
                 /       \
                 \       /
          _/\_/\_/\__  _/_/\_/\_/\_/\_/\_/\_/\_/\_/\_
          |   |   |  |( (  |  |  |  |  |  |  |  |  |  |
          |   |   |  | ) ) |  |  |  |  |  |  |  |  |  |
          |   |   |  |(_(  |  |  |  |  |  |  |  |  |  |
          |   |   |  |  |  |  |  |  |  |  |  |  |  |  |
          |   |   |  |  |  |  |  |  |  |  |  |  |  |  |
```

2 Variabler

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

2.1 Intro til variabler

2.1.1 Opprette variabler

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

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

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

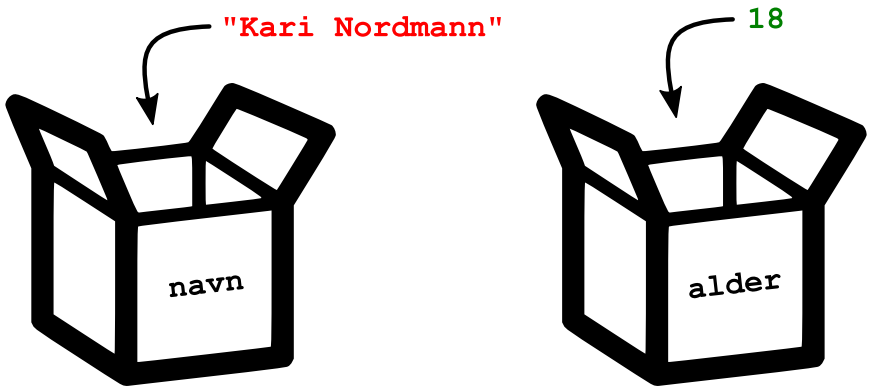
Det vi gjør når vi oppretter en variabel, er å fortelle datamaskinen at den skal huske på en bit med informasjon. I vårt eksempel ber vi den for eksempel huske på en person, og en alder. Informasjonen lagres i minnet på datamaskinen, og vi får et navn i programmet vårt vi kan bruke til å hente informasjonen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

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

2.1.2 Bruke variabler

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

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



Figur 1: Variabler kan beskrives som bokser der vi oppbevarer, eller lagrer, informasjon. Variablene har en *verdi*, som er boksens innhold, og et *navn*, som brukes til å referere til og finne riktig boks.

```
Hei på deg,
Kari Nordmann
```

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

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

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

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

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

Variabler er viktige nesten uansett hva slags program vi skal lage, og det er derfor

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

2.2 Flette variabler inn i tekst

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

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

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

```
Hei på deg Ola!
```

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

Eksempel: Adjektivhistorie

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

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

Før vi faktisk kan kjøre dette programmet må vi nå gå inn og definere variablene. Ellers blir Python forvirret når vi kjører programmet, fordi vi ber den flette inn variabler som ikke finnes!

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

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

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

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

2.3 Input

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

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

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

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

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

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

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

I eksempelkjøringen her i kompendiet ser du at brukeren skrev inn `Jonas`. Vi markerer det i eksempelet vårt med blått. Når du selv kjører dette programmet vil programmet stoppe opp, og du får se en markør i vinduet, slik at du kan skrive inn et svar.

Merk spesielt at vi skriver

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

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

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

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

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

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

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

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

2.3.1 Oppdatert adjektivhistorie

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

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

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

Etter å ha laget dette programmet og sett at det fungerer som det skal, så kan man nå kjøre det så bare terminalvinduet vises, og få en klassekamerat eller en annen venn til å fylle ut.

2.4 Utregninger

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

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

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

Her definerer vi at vi skal ha en variabel, `radius` som skal være lik `1.1`. Vi må også definere tallet π, siden Python ikke kjenner til det av seg selv. Når vi har gjort dette regner vi ut volumet til fotballen ved å skrive `volum =4*pi*radius**3/3`. Vi skriver `*` for å gange (asterisk/stjerne) og `**` for *opphøyd i*. Altså er `volum` lik

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

Merk at vi her ikke har noen enheter. Grunnen til det er at Python ikke har noen innebygd måte å håndtere dem, på samme måte som kalkulatorer ikke har noen måte å håndtere dem.

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

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

```
print(volum)
```

```
5.5724533333333355
```

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

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

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

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

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

Når vi skriver kodelinjen

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

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

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

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

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

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

Dette kan du prøve selv:

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

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

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

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

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

2.5 Matematiske operasjoner

Vi har alle de grunnleggende matematiske operasjonene tilgjengelig i Python. Merk spesielt at for potens skriver vi `**`, 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.

Innebygde Operasjoner

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

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

```
1 from math import sqrt
```

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

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

```
9.0
```

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

```
1 from math import *
```

Tabellen under lister noen få av de matematiske funksjonene og operasjonene vi kan importere. Alle disse finnes i `math`, men også i `pylab`, som er et bibliotek vi vil neste uke, det kommer vi tilbake til senere.

Kan importeres

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

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

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

```
3.141592653589793
2.718281828459045
```

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

2.5.1 Rekkefølge og prioritet

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

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

```
5.0
3.5
```

2.5.2 Oppdatere variabler

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

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

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

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

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

```
26450
```

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

På 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.

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

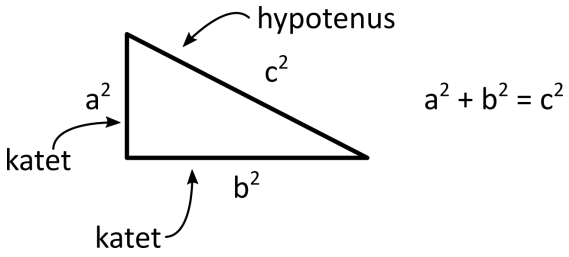
```
pris = 499
pris *= 0.7
print(f'Ny pris er {pris:.0f} kr')
```

```
Ny pris er 349 kr
```

2.6 Input og regning

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

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



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

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

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

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

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

```
7 + 8 = 15
```

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

Om vi derimot endrer hvordan vi definerer variablene til

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

```
7 + 8 = 78
```

Her blir jo svaret helt feil! Dette er fordi vi her lager to *tekstvariabler*, istedet for tallvariabler. 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.

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

Dette kan vi gjøre ved å skrive for eksempel

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

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

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

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

Så vi kan gå tilbake og oppdatere Pytagorasprogrammet vårt så det fungerer som det skal. Da må vi spesifisere at input skal være desimaltall, ved å legge til `float()`:

```python
from math import sqrt

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

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

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

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

3 Løkker

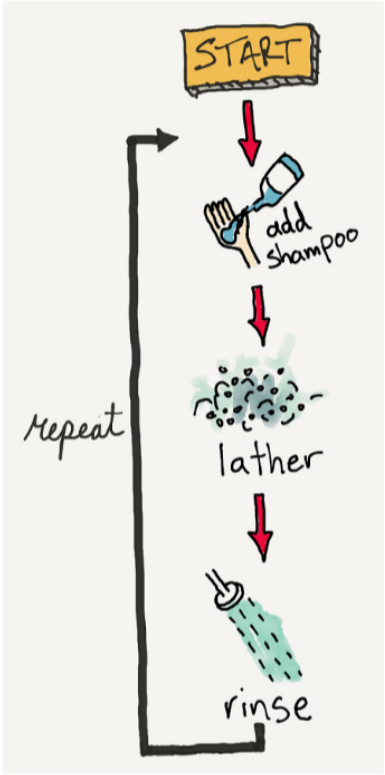
Løkker brukes for å gjenta en prosess mange ganger. Dette høres kanskje ikke så nyttig ut, men er det av de viktigste konseptene innen programmering. Den ene grunnen er at vi kan spare oss selv arbeid, ved å skrive en kort kode som datamaskinen gjennomfører mange ganger, istedenfor å måtte skrive ut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

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

3.1 Historie: Shampoo-algoritmen

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

- Lather



- Rinse
- Repeat

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

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

Sjampoalgoritmen er blitt en populær måte å introdusere konseptet algoritme, eller løkker som handler om gjentagelser. Eller bare som et eksempel på at man må være litt forsiktig når man programmerer, da en datamaskin gjerne følger instruksjoner bokstavelig. Det brukes også gjerne som en spøk. Selve sjampoalgoritmen er så velkjent i engelsktalende land at begrepet "rinse-repeat" brukes gjerne som et

idiom på å gjenta noe, selv om det ikke har noe med *skylling* å gjøre i det heletatt.

3.2 Frakoblet aktivitet: Navigasjon

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

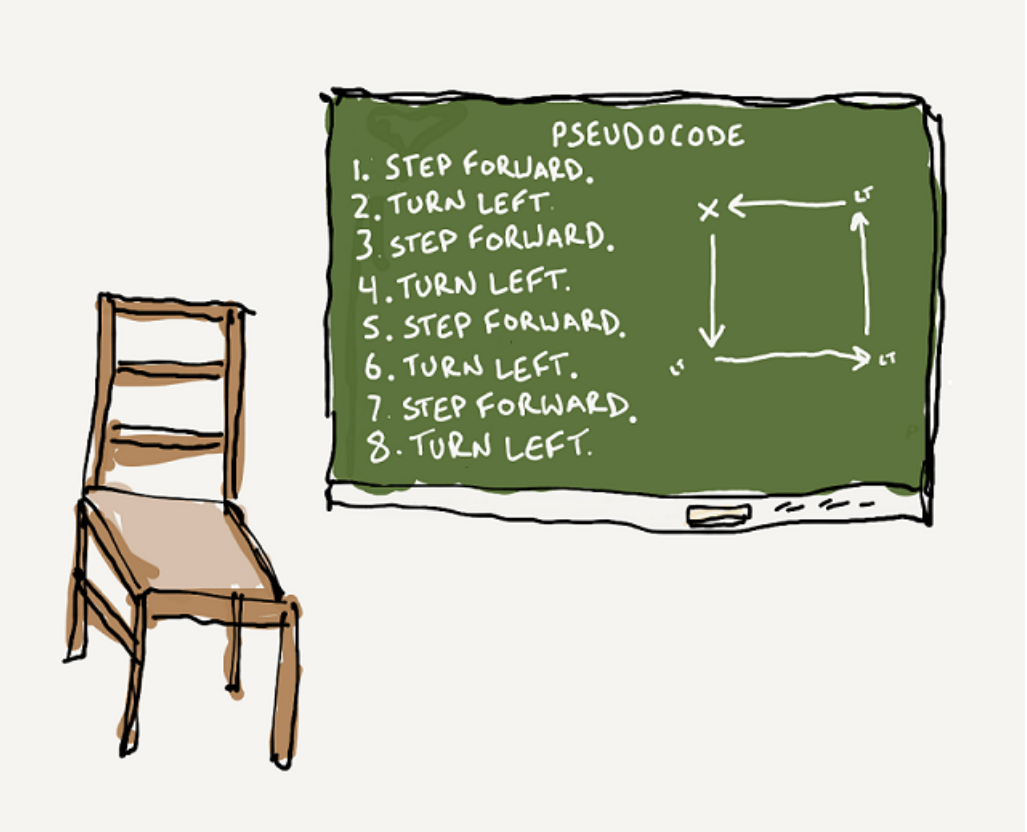
Sett en stol eller et annet objekt på gulvet med litt god plass rundt seg. Stell deg så litt bak og til siden for objektet.

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

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

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

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



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

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

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

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

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

3.3 Løkke over tallrekker

Kanskje de aller enkleste løkkene i Python er også kanskje de viktigste i matematikken, nemlig å løkke over en tallrekke. På englesk kalles en tallrekke for en "range", og vi bruker derfor funksjonen `range()` for å løkke over en tallrekke.

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

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

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

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

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

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

> for hvert tall i tallrekka fra og med 1 til 11, gjør følgende:

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

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

3.3.1 Eksempel: Kvadrattall og kubikktall

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

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

```
1
4
9
16
25
```

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

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

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

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

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

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

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

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

3.3.2 Sum av en tallrekke

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

```
total = 0

for tall in range(1, 11):
    total += tall

```

```
6 print(total)
```

```
55
```

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

3.3.3 Eksempel: Fakultet

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

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

Altså ganger vi tallrekka nedover til og med 1.

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

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

```
5! = 120
```

Hvis vi endrer `n` til å være lik `10` får vi i stedet:

```
10! = 3628800
```

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

Fakultet er viktig i kombinatorikk, fordi $n!$ er antall måter vi kan arrangere n ting. Ta for eksempel en vanlig kortstokk, som består av 52 kort (vi ser bortifra jokere).

Hvert kort er unikt, så det er 52! måter en kortstokk kan være arrangert på. Ved å bruke programmet vi lagde kan vi se at dette svarer til

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

et helt enormt tall! Om vi endrer `print`-kommandoen vår til `{total:g}`, så vil den skrive ut store tall på standardformat. Da kan vi sjekke ouput for `n = 52`:

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

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

3.3.4 Finkontrol på tallrekka

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

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

Isåfall tolkes dette som tallrekka

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

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

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

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

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

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

3.3.5 Eksempel: Renteberegninger

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

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

La oss si at Eirik får en årlig rente på 3.45 % på en BSU konto. Om han setter inn sine 10 000 kroner, hvordan vil disse vokse over tid? Dette gjøres enklest med en løkke, fordi vi for vært år skal gjenta de samme stegene: Først beregne rente, og så skrive ut fortjenesten.

```
1 penger = 10000
2 rente = 1.0345
3
4 for år in range(1, 11):
5     penger *= rente
6     print(f'Etter {år} år er det {penger} kr.')
```

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

Programmet fungerer helt fint, men selve utskriften blir rotete, fordi vi får med masse desimaler som ikke er viktige. Vi kan runde av til nærmeste desimal ved å skrive `penger:.0f` i f-printen vår, her betyr `.0` at vi skal ha 0 desimaler, og `f` at

det er en *float*, altså desimaltall. I tilegg skriver vi `{år:2}`, slik at antall år tar samme bredde, og tallene havner pent under hverandre.

```
penger = 10000
rente = 1.0345

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

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

Dette eksempelet viser hvor god fortjeneste man kan ha på sparing, spesielt når man tenker på rentesrente. Man kan lage en mer utvidet oppgave, eller et helt opplegg på dette der man for eksempel kan se på forskjellige måter å spare på. Eller man kan se på forskjellige lån, og hvor lang tid det tar å nedbetale dem med faste avdrag. Her kan man f.eks se på hvor forferdelige forbrukslån kan være.

4 Betingelser og logikk

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

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

4.1 Å skrive betingelser i Python

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

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

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

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

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

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

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

Det betyr at vi sammenligner to ting, `svar` og `'ja'`. Dersom de to er like er det sant, hvis de er ulike er det falskt. Vi bruker to likhetstegn (`==`) fordi ett likhetstegn er holdt av til å definere variabler. Merk at brukeren her må svare nøyaktig "ja" for at betingelsen skal stemme. Vi kan sjekke for flere betingelser ved å skrive `or`, altså "eller":

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

4.1.1 Hvis-ellers

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

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

if svar == 'Ja' or svar == 'ja':
    print('Hørt om veterinæren som var dyr i drift?')
else:
    print('Neivel, din surpomp')
```

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

4.1.2 Logiske operatorer

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

```
alder = int(input('Skriv inn et tall'))

if alder >= 0:
    print('Tallet er større en eller lik 0')
else:
    print('Tallet er negativt')
```

Her bruker vi altså `>=` for å sjekke om tallet er større eller lik 0.

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

Vanlige betingelser

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

Merk at hvordan *større enn* og *mindre enn* tolkes avhengig av hva slags type variabler vi snakker om. For tall er det jo ganske greit, men hva med for eksempel tekst? Her vil Python bruke den alfabetiske sorteringen, slik at for to tekststrenger vil `a > b` være sant hvis `a` ville blitt sortert før `b` om vi sorterterte dem alfabetisk. Dette er viktig med tanke på `input`, for om vi skal sammenligne dem som tall må vi gjøre dem om til tallvariabler.

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

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

4.2 Frakoblet: Rødt lys, grønt lys

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

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

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

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

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

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

Etter man har lekt leken et par ganger bør man reflektere tilbake på begrepsbruken. Her bør man gjerne knytte opp de norske begrepene man har brukt, men de engelske begrepene og eventuelt Python.

4.3 Mer enn to utfall

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

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

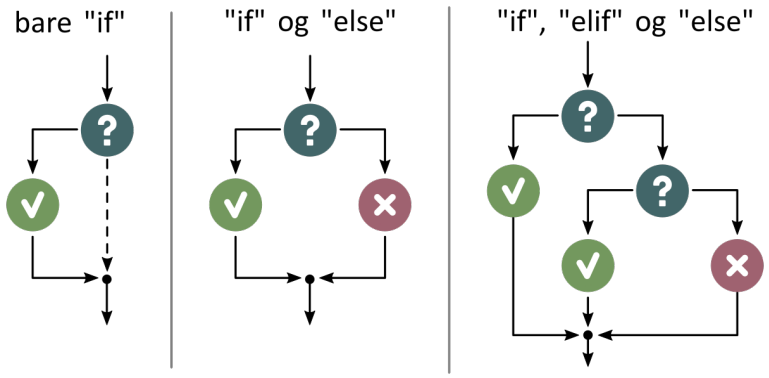
```python
poeng_a = 290
poeng_b = 320

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

Her sjekkes først den første betingelsen, hvis den er sann kjøres koden i den blokka og resten hoppes over. Hvis den første blokka er falsk, så sjekkes den neste, og så videre. Til slutt, hvis ingen andre blokker har kjørt, så kjøres *else*-blokka. Uansett så vil kun ett av utfallene skje.

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

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



Figur 3: Tre forskjellige typer betingelser. Til venstre finner vi en hvis-så setning, om betingelsen ikke er sann hopper vi bare over hele greia. I midten er en hvis-ellers, her gjør vi to ulike ting avhengig av betingelsen. Til slutt har vi en hvis-ellers hvis-ellers, denne kan bestå av så mange ulike forgreninger vi måtte ønsker.

4.4 Å bryte ut av løkker

Noen ganger har vi lyst til å avbryte løkker før de er ferdige, og det har python heldigvis lagt til funksjonalitet for. Måten vi gjør det på er ved å bruke `break` nøkkelordet i python. For å forstå hvordan det funker, tar vi et eksempel hvor vi vil printe ut alle kvadrattallene mindre enn 1000.

```
for tall in range(1, 1001):
    kvadrat = tall**2
    if kvadrat >= 1000:
        break
    print(kvadrat)
```

```
1
2
4
16
...
961
```

Her ser vi at løkken oppfører seg som vanlig, frem til `tall =32`. Når det skjer så vil `kvadrat` være lik `1024`, uttrykket i hvis-betingelsen er sant. Dermed blir `break` uttrykket kjørt av python og løkka avbrytes.

Et eksempel som illustrerer hvor dette kan være nyttig i matematikken er om vi skal undersøke vekst av rekker. La oss se på renteeksempelet fra løkkekapittelet igjen. Eirik får altså en årlig rente på 3.45 % på en BSU og setter inn 10 000 kroner. Nå lurer Eirik på om han har over 12 000 kroner før det har gått 10 år, og i såfall hvor mange år tok det. Her har vi altså et problem hvor `for-break` mønsteret er kjempenyttig, som vi ser av koden under

```
penger = 10000
rente = 1.0345
ventetid = 10

for år in range(1, ventetid + 1):
    penger *= rente
    if penger > 12000:
        print(f'Det tok {år} år før Eirik har 12 000,-')
        break
```

```
Det tok 6 år før Eirik har 12 000,-
```

La oss nå si at Eirik ville ha 15 000 kroner, da endrer vi koden slik

```
penger = 10000
rente = 1.0345
ventetid = 10

for år in range(1, ventetid + 1):
    penger *= rente
    if penger > 15000:
        print(f'Det tok {år} år før Eirik har 15 000,-')
        break
```

Her ser vi et problem, hvis Eirik ikke får 15 000 kroner på 10 år så får vi ingen utskrift. Heldigvis er det en løsning for dette i python, nemlig `for-break-else` mønsteret. I python kan vi faktisk ha en `else` kommando etter en `for`-løkke.

```
penger = 10000
rente = 1.0345
ventetid = 10

for år in range(1, ventetid + 1):
    penger *= rente
    if penger > 15000:
        print(f'Det tok {år} år før Eirik har 15 000,-')
        break
else:
    print(f'Eirik fikk ikke mer enn {penger:.2f} kroner på
        {ventetid} år')
```

Ved første øyekast virker kanskje dette litt pussig. Når er det *else* blokka blir kjørt? Den kjører kun dersom `for` løkka fikk løkket seg ferdig uten å bli avbrudt av en break. Utskriften fra koden over blir altså:

```
Eirik fikk ikke mer enn 14037.99 kroner på 10 år
```

Den beste måten å bli komfortabel med denne `for-break-else` strukturen er å prøve selv. Du kan for eksempel bytte ut verdiene for ventetid og rente i koden over og se hva slags utskrift du får.

5 Sammensatte eksempler

Nå har vi dekket alt vi skal av programmeringsgrunnlaget. Under følger to sammensatte eksempler, som viser hvordan man kan kombinere alt dere har lært til å løse kompliserte og sammensatte problemstillinger. Kapittelet etter dette har oppgaver det er lurt å prøve seg på først.

Disse eksemplene er fine for å gå igjennom og løse som grupper, eller tilogmed som hele klassen. Der læreren skriver kode på storskjerm etter innspill fra elevene.

5.1 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 betingelser 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.

5.1.1 FizzBuzz Frakoblet

La oss først forklare hvordan man leker FizzBuzz helt frakoblet fra datamaskin. Leken fungerer best med en litt større gruppe. Leken går ut på at man skal telle oppover som en gruppe. Så en person starter og sier 1, deretter sier neste i løkka 2, så sier nestemann 3 og så videre. Utfordringen kommer av at 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*.

Etterhvert som man teller oppover bør læreren følge med på at alle gjør riktig. Om noen bommer, eller bruker for lang tid, så avslutter man, og begynner fra starten.

Målet er å komme til så høye tall som mulig uten å gjøre feil, og det skal gjerne gå litt fort! Eventuelt kan man gjøre det til en konkurranse, der hver gang noen gjør feil, så går de ut.

Om du velger å leke FizzBuzz i ditt klasserom, prøv å gi leken litt ekstra piff, ved å insistere på at elevene er litt fremoverlent og på hugget. Det fører til ekstra engasjement og energi.

5.1.2 Å kode opp FizzBuzz

Nå ønsker vi å gå over til datamaskin, og kode opp FizzBuzz. Med dette mener vi at vi ønsker å lage et program som skriver ut det man ville sagt i leken. Vi skal altså lage et program som teller oppover, 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 20:

```
for tall in range(1, 21):
    print(tall)
```

```
1
2
3
...
20
```

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

- 7 delt på 2 gir en rest på 1. Derfor blir `7%2` lik 1.
- 6 delt på 3 gir ingen rest. Derfor blir `6%3` lik 0.
- 11 delt på 4 gir en rest på 3. Derfor blir `11%4` lik 3.

Å si at et tall er delelig med et annet, er det samme som å si at resten er 0. Vi kan derfor kombinere modulo med en hvis-betingelse, for å sjekke om hvert tall i rekka vår er delelig med 3.

```python
for tall in range(1, 21):
    # Er tallet delelig med 3?
    if tall % 3 == 0:
        print('Fizz')
    else:
        print(tall)
```

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

Om vi kjører programmet og sjekker ser vi at det fungerer som forventet.

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

```python
for tall in range(1, 21):
    # Er tallet delelig på 3?
    if tall % 3 == 0:
        print('Fizz')

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

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

```
1
2
Fizz
4
Buzz
```

```
Fizz
7
...
```

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

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

```
for tall in range(1, 21):
    # 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 $3 \times 5 = 15$.

Nå har vi egentlig løst oppgaven, og lagd et program som leker FizzBuzz helt perfekt. Om vi ønsker å gå enda lenger kan vi for eksempel legge inn at den spør brukeren hvor høyt den skal gå, isteden for å stoppe på 20. Eller så kan vi legge på enda flere regler, og her setter kreativiteten grensene. Kanskje vi for eksempel skal gå opp til 100, og så ned igjen? Eller hva med å si at alle tall i ti-gangen skal hoppes helt over?

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 betingelser. 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.

5.2 Finne primtall

En annen fin oppgave som dekker mye av det som er dekket i dette kompendiet er å lage et program som sjekker om et gitt tall er et primtall eller ikke.

5.2.1 Frakoblet

Før man skal igang med å kode opp et et program som finner primtall er det lurt å ta litt tid til å repetere hva primtall er, og hvordan vi kan finne ut om et tall er et primtall eller ikke for hånd.

Definisjonen på et primtall er et heltall som kun kan deles på 1 eller seg selv. Derimot er ikke 1 et primtall.

Nå kan det være lurt å stille spørsmålet: Hvis jeg gir deg ett tall, for eksempel 13, hvordan sjekker du om det er et primtall? Ut ifra definisjonen må vi sjekke om 13 er delelig med noen av tallene i rekka

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

Så for hvert tall prøver vi å dele:

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

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

Om vi prøver med 15

$$15/2 = 7.5$$
$$15/3 = 5$$

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

5.2.2 Programmert opp

La oss nå lage et program som spør brukeren om et tall n, og sier ifra om tallet er et primtall eller ikke.

Først spør vi om n, da må vi huske å konvertere til en tallvariabel

```
n = int(input("Hvilket tall vil du sjekke?"))
```

Siden 1 er litt spesiell, så burde vi sjekke denne spesielt:

```
if n == 1:
    print("1 er ikke et primtall.")
else:
    ...
```

Vi skal nå skrive inn koden for alle andre tall i else-blokken.

Vi lager da løkka over tallene $2, 3, \ldots n-1$ ved å bruke range. Vi sjekker om et tall er delelig ved å sjekke resten med modulooperatoren, dette står beskrevet i det forrige eksempelet (FizzBuzz).

```
for d in range(2, n):
    if n % d == 0:
        print(f"{n} er ikke et primtall")
        print(f"(Det er f.eks delelig med {d})")
        break
```

Hvis vi finner et tall kandidaten n er delelig er det følgelig *ikke* et primtall, så da skriver vi det ut. Da kan vi også *avbryte* løkka, det har vi ikke dekket hvordan man gjør, men man kan gjøre det med kommandoen `break`. Ordet "break" betyr å bryte eller ødelegge, så vi sier at vi *bryter* løkka.

Da gjenstår det bare én ting. Å skrive ut at tallet *er* et primtall. Dette skal vi gjøre bare hvis løkka fullfører som normalt, uten å finne noen divisorer. Dette kan vi gjøre på et par forskjellige måter, men den letteste er kanskje det at en for-løkke kan ha en tilhørende *else*-blokk. Sånn som dette:

```
for d in range(2, n):
    if n % d == 0:
        print(f"{n} er ikke et primtall")
        print(f"(Det er f.eks delelig med {d})")
        break

else:
    print(f"{n} er et primtall!")
```

Her må man passe litt på innrykkene, fordi `else`-blokken hører til `for`-løkka, ikke hvis-betingelsen inne i løkka.

La oss repetere nøyaktig hva denne `for-break-else` strukturen betyr. Om vi tenker tilbake på hvordan vi formulerte algoritmen for å finne et primtall, så sier vi at *hvis vi finner en divisor* så er tallet ikke primtall, og *ellers* så er den det. Måten Python tolker en `else` etter en for-løkke er derfor at dersom løkka er brutt med `break`, så skal betingelsen ikke slå inn, men om løkka fullfører som normalt, så har vi ikke funnet noe, så da skal *ellers* betingelsen slå inn.

Da er hele programmet som følger, med litt ekstra kommentarer

```
# Spør brukeren om et tall
n = int(input("Hvilket tall vil du sjekke? "))

# Behandler 1 spesielt, for den er litt spesiell
if n == 1:
    print("1 er ikke et primtall.")

# Alle andre tall
else:
    # Løkk over 2, ..., n-1
    for d in range(2, n):
        # Sjekk om n er delelig med tallet
        if n % d == 0:
            print(f"{n} er ikke et primtall")
            print(f"(Det er f.eks delelig med {d})")
            # Vi er ferdig, så bryt løkka
```

```
17              break
18
19      # Løkka er ferdig, men fant ingen divisorer
20      else:
21          print(f"{n} er et primtall!")
```

Dette programmet kan nå sjekke om et hvilket som helst (positivt) heltall n er et primtall. Det var litt jobb å kode det opp, men det var god repetisjon i hva primtall er, og man kan lære mye av en slik øvelse. Det å sjekke om et tall er et primtall er også en øvelse som tar lang tid for hånd, så ved å ha laget dette programmet kan vi spare masse tid om vi noen gang trenger å sjekke store tall.

Er for eksempel 40193 et primtall? Det er ikke bare å finne ut av! Eller, med programmet vårt så er det jo det:

```
Hvilket tall vil du sjekke? 40193
40193 er et primtall!
```

Utvidelser

Man kan nå også gå enda lenger om man ønsker å gjøre forbedringer. Trenger vi for eksempel å sjekke alle tall oppover rekka?

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

Nei, det gjør vi ikke. For det første er det unødvendig å sjekke alle partalene, fordi et tall som er delelig på et partall må også være delelig på 2, som vi sjekke 2, trenger vi ikke sjekke alle de andre partallene: 4, 6, 8, osv. Dette halverer effektivt antall tall vi trenger å sjekke.

Hvor viktig er det å halvere antall tall vi må sjekke? Når vi jobber for hånd kan dette spare oss masse tid, fordi å sjekke hvert enkelt tall tar lang tid. Datamaskinen jobber veldig raskt, så for forholdsvis små tall, kanskje opptill et par millioner, så vil datamaskinen være så raskt at vi sparer ingen tid ved å halvere antall steg. Om vi ønsker å sjekke veldig store tall derimot, som f.eks

$$141245151513,$$

så vil det ta lang tid, selv for en datamaskin, og vi kan være mer effektive.

Tilsvarende trenger vi ikke å sjekke alle tall helt opp til $n - 1$. Det holder faktisk å sjekke alle tall opp til $\sqrt{n}$. Dette vil spare oss veldig masse tid, fordi det gjør at vi må sjekke *langt* færre tall. Hvorfor trenger vi bare å sjekke opp til $\sqrt{n}$. Det kan dere jo bruke litt tid på å tenke på selv.

Algoritmen vi har laget i Python her, er bare én mulig måte å finne primtall på, og det finnes mange andre. Den vi har brukt er kanskje den mest intuitive og mest "rett frem". Andre metoder kan gjerne være raskere, eller bedre på andre måter. For eksempel kan man lage noe som heter en *primtallssil*. Men det dekker vi ikke her og nå.

5.3 Eksempel: Gjettespillet Over/Under

Over/under er en enkel gjettelek to personer kan leke, som også egner seg godt til å programmere.

Frakoblet

Over/under lekes ved at man settes sammen i par. Person A tenker på et tilfeldig tall mellom 1 og 1000 og skriver det ned på et ark, så den andre personen ikke kan se det.

Deretter skal person B prøve å gjette seg frem til tillet. Etter hvert forslag B gir, så skal person A si om gjettet var over, under, eller helt riktig. På denne måten får B mer og mer informasjon for hvert gjett, og kan peile seg inn til riktig svar.

Etter at person B gjetter helt riktig tall, bytter man roller. Denne gangen skal B finne på et tall, og A gjette seg frem til det. Målet er å komme frem til riktig tall på færrest mulig gjett.

Om man leker dette i klasserommet kan man for eksempel la elevene leke tre runder hver. For hver runde bør de skrive ned hvor mange gjett de brukte. Den eleven som klarte å få til alle tre rundene på færrest gjett totalt er vinneren.

På datamaskin

La oss nå prøve å kode opp dette spillet. Først må vi definere hvilket tall datamaskinen tenker på. Her kan vi velge hva vi vil:

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

Merk programmet definerer et fasittall, men skriver det ikke ut. Fasiten er altså hemmelig for spilleren, akkurat som når man velger et tall og skriver det ned på en lapp som man ikke viser frem. Etter dette skriver vi ut instrukser til spilleren. (Når vi skriver `print()` uten noe beskjed i parentesene så printer vi en blank linje).

Vi skal nå gjenta den samme prosessen mange ganger: nemlig la brukeren gjette.

For å gjenta noe mange ganger bruker vi en *løkke*. Når vi bruker en `for`-løkke må vi si hvor mange ganger vi løkker, så la oss for eksempel gi brukeren 100 gjett. For hver løkke må vi spørre brukeren om et gjett med `input`, og huske å konvertere svaret fra en tekstvariabel til en tallvariabel med `int`:

```
for gjett in range(1, 101):
    svar = int(input("Gjett: "))
```

Etter brukeren gir oss et svar ønsker vi nå å sjekke om det er: for lavt, for høyt, eller helt riktig. Dette kan vi gjøre med en hvis-betingelse. Først kan vi sjekke mindre-enn og større-enn:

```
if svar > fasitsvar:
    print(f"Ditt gjett på {svar} er for høyt.")
elif svar < fasitsvar:
    print(f"Ditt gjett på {svar} er for lavt.")
```

Til slutt vil vi sjekke om de treffer helt riktig. I såfall ønsker vi å skrive ut hvor mange gjett de har brukt totalt sett. Vi ønsker også isåfall å bruke `break`, for da bryter vi løkka:

```
elif svar == fasitsvar:
    print("Helt riktig!")
    print(f"Du brukte {gjett} gjettinger!")
    break
```

Hva skjer om brukeren bruker opp alle 100 gjettene sine? Da er løkka ferdig, og de får ikke lov til å gjette mer. Isåfall bør vi kanskje skrive ut en beskjed. Det kan vi gjøre ved å ha en `else`-blokk som hører til løkka vår:

```
else:
    print(f"Der har du prøvd 100 ganger og gått tom for
       gjett! Riktig svar er {fasitsvar}")
```

Her bruker vi altså en for-else, eller på Norsk, en for-ellers løkke. Det som står i else skjer bare dersom programmet kommer igjennom hele løkka uten å treffe på en `break` kommando. I vårt tilfelle vil det si at brukeren *ikke* finner riktig svar før han/hun har gått tom for gjett.

Hele programmet blir nå

```
fasitsvar = 45
```

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

for gjett in range(1, 101):
    svar = int(input("Gjett: "))

    if svar > fasitsvar:
        print(f"Ditt gjett på {svar} er for høyt.")
    elif svar < fasitsvar:
        print(f"Ditt gjett på {svar} er for lavt.")
    elif svar == fasitsvar:
        print("Helt riktig!")
        print(f"Du brukte {gjett} gjettinger!")
        break
else:
    print(f"Der har du prøvd 100 ganger og gått tom for
       gjett! Riktig svar er {fasitsvar}")
```

En kjøring av programmet gir nå for eksempel:

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

Gjett: 400
Ditt gjett på 400 er for lavt.
Gjett: 700
Ditt gjett på 700 er for høyt.
Gjett: 600
Ditt gjett på 600 er for høyt.
Gjett: 500
Ditt gjett på 500 er for høyt.
Gjett: 450
Ditt gjett på 450 er for lavt.
Gjett: 475
Ditt gjett på 475 er for høyt.
Gjett: 465
Ditt gjett på 465 er for høyt.
Gjett: 455
Ditt gjett på 455 er for lavt.
Gjett: 462
```

```
Ditt gjett på 462 er for høyt.
Gjett: 459
Ditt gjett på 459 er for høyt.
Gjett: 457
Ditt gjett på 457 er for høyt.
Gjett: 455
Ditt gjett på 455 er for lavt.
Gjett: 456
Helt riktig!
Du brukte 13 gjettinger!
```

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.

5.3.1 Bonusutvidelse: La datamaskinen tenke på et tilfeldig tall

Et mulig problem med eksempelet over er at vi definerer fasittallet på starten. Det betyr at hvis du ser på koden så får du se fasiten, og da er det jo ikke noe poeng

å gjette! Det er derfor lurt å gjemme koden når du spiller spillet.

En annen løsning er å la datamaskinen tenke på et helt tilfeldig tall. Det finnes heldigvis en ferdig funksjon i python som trekker et tilfeldig tall, `randint`. `randint` står for **rand**om **int**eger altså "tilfeldig heltall". Den tar inn to tall som input og returnerer et tilfeldig tall mellom de to tallene. For å bruke `randint` må vi importere den fra en pakke som heter `random`. Eksempel:

```
from random import randint
tilfeldigtall = randint(1, 1000)
print(tilfeldigtall)
```

Hvis du kjører dette programmet mange ganger ser du at `tilfeldigtall` har forskjellig verdi mellom 0 og 1000 hver gang. Dette kan vi bruke til å la datamaskinen "tenke på" et tilfeldig `fasittall` i gjetteleken hver gang vi spiller spillet. Den modifiserte koden blir som følger:

```
from random import randint

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

for gjett in range(1, 101):
    svar = int(input("Gjett: "))

    if svar > fasitsvar:
        print(f"Ditt gjett på {svar} er for høyt.")
    elif svar < fasitsvar:
        print(f"Ditt gjett på {svar} er for lavt.")
    elif svar == fasitsvar:
        print("Helt riktig!")
        print(f"Du brukte {gjett} gjettinger!")
        break
else:
    print(f"Der har du prøvd 100 ganger og gått tom for
       gjett! Riktig svar er {fasitsvar}")
```

Nå vil selv ikke personen som skrev koden vite hva fasiten er, og det vil være ny fasit hver gang du spiller spillet.