En innføring i programmering med Python

Vi skal nå sette igang med å programmere. Fordelt over to notebooks går vi igjennom en rekke grunnleggende konsepter i Python. Vi viser en del eksempler og eksempeloppgaver iløpet av veien, men i tilegg finnes det en til notebook kun med oppgaver.

Et eksempelprogram

Før vi begynner å gå igjennom detaljert ønsker vi å vise frem et eksempelprogram, vi kommer til å dekke alt i dette programmet i detalj senere, så det er ikke viktig at dere skjønner alt som skjer - det er mest tenkt for å vise hvordan et program er bygget opp, og hvordan vi kan analysere det. Vi viser koden først, så forklarer vi litt hva som skjer under.

In [1]:

```
# Importer nødvendig funksjonalitet
from pylab import pi

# Sett radius
radius = 11

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

# Skriv ut resultater
print("En fotball med radius på {} cm".format(radius))
print("Har en omkrets på {:.1f} cm".format(omkrets))
print("Et overflateareal på {:.1f} cm^2".format(overflate))
print("Og et volum på {:.1f} cm^3".format(volum))
```

```
En fotball med radius på 11 cm
Har en omkrets på 69.1 cm
Et overflateareal på 1520.5 cm^2
Og et volum på 5575.3 cm^3
```

Det vi her kaller et "program" er er sett med kodelinjer, eller *statements* som de kalles på engelsk. Når vi kjører koden, så vil Python gå igjennom linje for linje og tolke det som står. Hver kodelinje er altså en bestemt instruks.

Den første linjen vi skriver starter med et `#` symbol, slike linjer kaller vi kommentarer, og disse linjene er egentlig ikke kode, de er bare beskrivelser vi legger til for at koden vår skal være lettere å lese for andre. Alle linjer i programmet som begynner med `#` er kun beskrivende, og gjør ingen forskjell for selve utførelsen av programmet.

Den første kodelinjen som 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 kulen vår til å være 11, med linjen `radius = 11`. Her *definerer* vi en radiusen som 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 ballen. 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 dette for oss.

Til slutt har vi 4 kodelinjer som bruker funksjonen `print`. I Python skriver `print` ut noe til skjermen (ikke til en printer), og målet er at vi skal fortelle brukeren informasjonen vi har regnet oss frem til. Her ser `print`-kommandoen kanskje litt komplisert ut, men det kan gjøres enklere, dette kommer vi tilbake til.

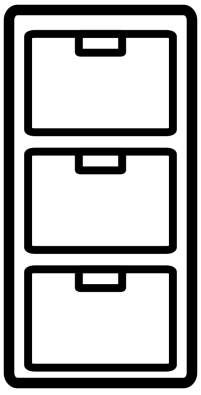
Kunsten å analysere kode

Det kan være noe vanskelig å lese kode når man er helt fersk til programmering, det kan fort bli litt mye og uoversiktelig rett og slett. Her er det viktig å lese programmet linje for linje, og prøve å forstå stegene. Et program har også en viss baktanke, du kan stille spørsmålet: *hva er det dette programmet prøver å gjøre?*. Med det i tankene kan du nå prøve å skjønne de ulike stegene programmet tar for å nå dette målet. Ved å ta det steg for steg blir det forhåpentligvis lettere å skjønne helheten.

Variabler

Vi er nå klare for å begynne å gå igjennom en del grunnleggende elementer i programmering og Python. Vi begynner med *variabler*. Variabler er noe av det mest fundamentale i programmering, for variabler er slik vi kan lagre og gjenbruke informasjon i programmene våres. Når vi skrev linjen `radius = 11` i programmet vårt, sier vi at vi *oppretter* en variabel. Vi kan også si at vi *definerer* eller *lager* variabelen. På engelsk kalles denne operasjonen for *assignment*, som betyr å tildele. Det vi gjør når vi oppretter en variabel er å fortelle Python at den skal huske på en bit med informasjon. Informasjonen lagres i minnet på datamaskinen, og vi får et navn i programmet vårt vi kan bruke til å hente informasjonen ut igjen og bruke senere.

I vårt eksempel oppretter vi en variabel og gir den *navnet* `radius`, og gir den *verdien* `11`. Vi kan tenke på en variabel som en oppbevaringsboks. Verdien er innholdet i boksen, mens navnet er det vi skriver utenpå, så vi kan finne riktig boks når vi trenger den. Python tar seg av ansvaret med å sørge for at informasjonen arkiveres i minnet på en måte der den raskt kan hente det ut igjen når vi spør om den.



Når vi definerer en ny variabel vil Python lage en helt ny boks. Vi kan for eksempel skrive `navn = "Karl"`, og vi får en *tekstvariabel* som inneholder navnet "Karl". Det er ingen begresning for hvor mange ulike variabler vi kan bruke i programmene våres. Om vi lager en ny variabel med samme navn som en variabel som allerede eksisterer så overskriver vi den gamle

In [2]:

```
radius = 11
radius = 20
print(radius)
```

```
20
```

Når vi overskriver en variabel på denne måten kan vi tenke på det som at Python går inn i boksen og erstatter innholdet. Det gamle innholdet har nå ingen sted å være, og blir kastet. Dette er faktisk ikke helt konkret det som skjer i bakgrunnen, for det Python egentlig gjør er å opprette en *ny variabel* og gir denne navnet `radius`. Det forrige variabelen mister så navnet sitt. Når Python så ser at det finnes en variabel (en boks) på lageret uten noe navn så kastes denne i søppla - dene prossesen kalles *garbage collection*. For brukeren utgjør det liten forskjell i praksis at det opprettes en ny variabel og den gamle kastes. Når vi overskriver en variabel på denne måten kaller vi det gjerne for å *redefinere* variabelen, og det går fint ann å tenke på det som at Python går inn og endrer innholdet i boksen.

Bruk av `=` for å lage variabler

Det er veldig vanlig å bruke `=` for å opprette variabler i programmeringsspråk, dette tegnet kalles gjerne for *assignment operator*. Det kan føre til noe forvirrelse, fordi likhetstegnet har en veldig konkret betydning i matematikken, og her bruker vi symbolet med en annen betydning.

For å forstå hva `=` betyr i Python kan vi tenke på det som en pil som peker mot venstre. Så når vi skriver `radius = 11` er dette litt som om det hadde stått `radius ← 11`, som kan tolkes som at det skal opprettes en ny variabel med navn `radius`, som skal få verdien `11`. I Python betyr altså `=` operatoren at det på høyre side skal legges inn i variabelnavnet på venstre side.

Et tilfelle som illustrer dette godt er uttrykket `x = x + 10`. Fra et matematisk ståsted ser dette ganske fryktelig ut, ligningen sier jo at $0 = 10$! Men i Python er dette faktisk helt greit, som vi kan se:

In [3]:

```
x = 5
x = x + 10
print(x)
```

```
15
```

Det som skjer her er at vi først definerer at `x` skal være 5. Så sier vi at `x` skal være `x + 10`. Da tolker først Python det som står på høyre side, og her ser den `x + 10`, så den sjekker og finner at `x` er 5, som vil si at høyre side er `5 + 10` som blir `15`. Først når Python er helt ferdig med å tolke høyre side av likhetstegnet settes verdien inn i `x`, som da blir redefinert til å være 15. Det er kanskje lettere å tolke denne prossesen om vi skriver det som

```
x ← 5
x ← x + 10
```

La oss bare understreke at `←` ikke er et symbol Python kan tolke, dette er bare en tankehjelp vi bruker for å illustrere.

Denne operasjonen med å legge en verdi til en variabel er veldig nyttig i programmering, den er faktisk så nyttig at vi kan bruke en liten snarvei og skrive `x += 10` for å endre en verdi. Her er formene `x = x + 10` og `x += 10` helt ekvivalente og gir likt resultat, så det er bare å bruke den som føles mest intuitiv.

Variabler og Jupyter

I Python eksisterer variabler helt til det ikke er bruk for dem lenger, eller til programmet er ferdig. I et vanlig kodescript er det derfor ganske greit å holde styr på hvilke variabler som finnes og ikke. I Jupyter derimot, funkerer ting litt annerledes. Her eksisterer variablene så lenge notebooken kjører. Det betyr at om vi kjører en celle, så husker Python på variabelen for neste celle vi kjører. La oss prøve:

In [4]:

```
1 print(radius)
```

```
20
```

Dette betyr at vi bør tenkte på hele notebooken, og ikke hver enkelte kodecelle, som et program. Vi kan faktisk laste ned dette programmet ved å gå på `File` > `Download as` > `Python`, da får vi en kodefil som inkluderer all koden vi har i notebooken vår, og teksktcellene inkluderes som kommentarer.

En ulempe med det at cellene henger sammen er at man kan kjøre cellene i feil rekkefølge, og plutselig har man prøvd å bruke et resultat før det faktisk er regnet ut! Her er trikset alltid å kjøre cellene fra starten av notebooken og nedover. Om det blir rot kan man restarte hele notebooken og kjøre alle celler i rekkefølge, det gjør man ved å klikke på verktøylinja over `Kernel` > `Restart & Run all`. Om man vil restarte notebooken men ikke kjøre cellene har vi også `Kernel` > `Restart & Clear Output`, dette kan man for eksempel gjøre før man skal vise frem notebooken til noen andre, så man kan kjøre celle for celle mens de ser på.

Bruk av `print` funksjonen

Vi har allerede sett en del eksempler på bruk av `print`, funksjonen som skriver resultater ut til skjermen slik at vi kan lese dem. Her er `print` det første eksempelet vi har på en *funksjon*. I Python skrives alle funksjoner på formen `f(x)`, der `f` er navnet på funksjonen, og `x` er *input*. Vi kan sende variabler av alle typer inn til `print`, og den vil prøve å vise det på en fornuftig måte.

Om vi ønsker å skrive ut en beskjed må vi bruker fnutter (`"` eller `'`) på begge sider av teksten, slik at Python skjønner at det er en tekst, og ikke kode som skal tolkes. Dette kaller vi en *tekststreng*.

In [5]:

```
1 print("Hello, World!")
```

```
Hello, World!
```

Vi kan også kombinere en beskjed med en variabel som følger

In [6]:

```
name = "Ragnar"
print("Hei på deg {}!".format(name))
```

```
Hei på deg Ragnar!
```

Her skriver vi en beskjed, men vi skriver `{}` inne i tekststrengen. Du kan tenke på dette som at vi lager en grop i teksten som vi senere kan fylle inn med en gitt verdi, og dette gjør vi ved å skrive `.format(name)` etter strengen med det vi skal fylle inn.

Vi kan også skrive ut andre ting, som for eksempel tall:

In [7]:

```
print(radius)
print(overflate)
```

```
20
1520.53084433746
```

Vi ser at overflaten skrives ut med veldig mange desimaler, og det ser ikke så pent ut. Dette kan vi også kontrollere ved hjelp av `.format`.

In [8]:

```
print("{:.1f}".format(overflate))
```

```
1520.5
```

På samme måte som over bruker vi `{}` til å lage en "grop", men denne gangen skriver vi `:.1f` inne i parantesene for å spesifisere at vi ønsker bare én desimal (`.1`) og at vi skal skrive ut et desimaltall (`f` for *float*, vi kommer tilbake til hva dette betyr). Ved hjelp av `{}` og `.format` får vi full kontrol over det vi skriver ut.

Eksempeloppgave: Adjektivhistorie

En veldig enkel oppgave som kan engasjere elever er å lage en enkel adjektivhistorie. Her lager vi først en kort historie som en tekststreng, med slike groper (`{}`), der adjektivene skal fylles inn, så kan vi definere en rekke adjektiv, og fylle inn når vi printer ut. Når vi lager tekststrenger som går over én linje kan vi bruke tre fnutter på hver side, da kan strengen bestå av mange linjer.

In [9]:

```
adj1 = "rød"
adj2 = "bløt"
adj3 = "høy"
adj4 = "sint"
adj5 = "bråkete"
adj6 = "seig"

historie = """En {} høstdag i oktober skulle den {}e klasse 8B
på telttur. Alle de {}e elevene hadde med
seg soveposer, varmt tøy og en {} sekk med {} mat.
Alle elevene gledet seg til denne {}e turen."""

print(historie.format(adj1, adj2, adj3, adj4, adj5, adj6))
```

```
En rød høstdag i oktober skulle den bløte klasse 8B
på telttur. Alle de høye elevene hadde med
seg soveposer, varmt tøy og en sint sekk med bråkete mat.
Alle elevene gledet seg til denne seige turen.
```

Dette er en enkel og engasjerende oppgave som viser litt om enkle Python kommandoer som `print` og opprettelse av variabler. Etterhvert skal vi lære om løkker og lister som kan brukes til å forbedre dette programmet. Vi kan også la adjektivene gis når vi kjører programmet. Dette kan vi gjøre ved å bruke `input`, som spør om en input fra brukeren

In [10]:

```
adj1 = input("Skriv et adjektiv: ")
adj2 = input("...og så et til: ")

historie = "Det {}e toget kjørte i full fart mot den {}e byen."
print(historie.format(adj1, adj2))
```

```
Skriv et adjektiv: rød
...og så et til: søt
Det røde toget kjørte i full fart mot den søte byen.
```

Å regne i Python

Vi har allerede sett litt eksempler på hvordan vi kan gjøre beregninger og matematikk i Python. La oss gå igjennom dette i litt mer detalj.

Opprette variabler som resultat av utregninger

Når vi skriver en kodelinje som

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

Oppretter vi en variabel som heter `volum`, men hva blir verdien? Som vi forklarte tidliger vil Python først se på høyresiden og tolke den, og så putte resultatet inn i variabelen på venstre side. I dette tilfellet er høyre siden et matematisk uttrykk, så her vil Python først regne seg frem til en tallverdi som legges inn i variabelen. Vi kan altså tenke på dette som det skjer i to steg: først regner den ut svaret, så oppretter den variabelen.

En viktig konsekvens av denne prosessen er det er *resultatet* som lagres, ikke den matematiske formelen. Når variabelen er opprettet glemmer Python hvor tallet kom fra! Om vi først regner ut og oppretter `volum`, og deretter endrer verdien i `radius`, så vil **ikke** `volum` endre seg. Dette kan vi teste:

In [11]:

```
r = 1
volum = 4*pi*r**3/3
print("Radius: {} Volum: {:.1f}".format(r, volum))

r = 10
print("Radius: {} Volum: {:.1f}".format(r, volum))
```

```
Radius: 1 Volum: 4.2
Radius: 10 Volum: 4.2
```

Her ser vi at selv om vi har redefinert `r`, så endret ikke verdien til `volum` seg.

Eksempeloppgaver: Å leke datamaskin

For å bli vant med hvordan det å opprette variabler fungerer, og for å øve seg på å tenkte algoritmisk, kan vi lage små kodesnutter elevene skal lese og tolke som om de var datamaskinen og prøve å komme frem til hva sluttresultatet skal bli. Hvis elevene jobber i par eller grupper kan de også øve seg på å snakke kode ved å skulle forklare for hverandre hva som skjer i programmet, linje for linje. Til slutt kan man kjøre koden og se om man hadde rett:

In [12]:

```
x = 3
y = -3
z = x + y
z = z + x
x = x + y + z
```

Hvilken verdi har nå `x`?

In [13]:

```
x = 0
y = 0
z = 0
x += 5
y *= x
z = x + y
```

Hva blir `z` her?

Slike oppgaver er gode for å få elevene til å virkelig tenke igjennom at Python tolker koden linje for linje, og at når man tenker som en datamaskin lønner det seg å gjøre det samme. Slike oppgaver kan også utvides etterhvert som man lærer nye konsepter i programmering, da vi kan legge inn forskjellige type operasjoner.

Matematiske operasjoner i Python

Vi har sett noen eksempler på utregninger i Python, blant annet har vi sett at det er rett frem å bruke de vanlige matematiske operasjonene, vi må bare bruke de riktig symbolene. Her er en tabell over de vanligste:

Matematisk Operasjon	Python
Addisjon	`a + b`
Subtraksjon	`a - b`
Multiplikasjon	`a * b`
Divisjon	`a / b`
Potens	`a**b`
Heltallsdivisjon	`a // b`
Modulus	`a % b`
Kvadratrot	`sqrt(x)`
Logaritme	`log(x)`
10-er Logartime	`log10(x)`
2-er Logaritme	`log2(x)`
Sinus	`sin(x)`
e^x	`exp(x)`

Merk spesielt at `**` betyr potens, og ikke `^`, som er vanlig i noen andre programmeringsspråk. I Python betyr `^` noe annet.

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

In [14]:

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

print(a)
print(b)
```

```
5.0
3.5
```

En annen ting vi må nevne er at grunnleggende Python ikke har så mye støtte for matematikk, det betyr at funksjonene kvadratrot, logaritme, eksponentialfunksjon og lignende ikke er tilgjenglig. Vi kan derimot *importere* dem for å få tilgang på dem. Vi kan få tilgang på alt vi får bruk for ved én enkel linje.

In [15]:

```
from pylab import *
```

Her sier vi at vi ønsker å *importere* ekstra funksjonalitet fra en pakke som heter *pylab*. Her kunne vi enten lagd en liste over akkurat det vi trenger `from pylab import pi, sin, exp`, eller vi kan bare skrive `*`, som betyr "alt". Pylab-pakken inneholder masse funksjoner for matematikk, statistikk og tegning av grafer. Navnet

"pylab" kommer fra at den er laget for at Python skal få tilgang på funksjonalitet som ligner på matematikkprogrammet MatLab. I tilegg til matematiske operasjoner og funksjoner for plotting har pylab også noen matematiske konstanter, som π og e

In [16]:

```
print(pi)
print(e)
```

```
3.141592653589793
2.718281828459045
```

Feilmelding: NameError

Når vi skriver kode datamaskinen ikke forstår kræsjer programmet vårt, og vi får en feilmelding ut. Alle kodelinjer etter dette vil ikke bli kjørt, så når programmet vårt feiler på et punkt betyr det gjerne at hele programmet slutter å fungere. Noen feil vil Python til og med plukke opp på før det kommer til den kodelinja. Feilmeldingen som skrives ut prøver å fortelle oss to ting: hvor noe gikk galt, og hva som gikk galt.

Hvor det gikk galt betyr ofte hvilken linje i programmet. I små programmer er dette veldig enkelt, for vi har ikke så mange muligheter. Men om et program begynner å gå over mer enn én side kan det bli litt vanskelig å se akkurat hvilken linje det er snakk om, og da er *linjenummer* veldig nyttig. I Jupyter kan du skru på linjenummer øverst ved å gå på `View -> Toggle Line Numbers` . Dette er ofte ikke nødvendig i Jupyter, da vi kan dele koden vår over flere celler, men det er kjekt å kunne skru det på når det trengs.

Den andre biten med informasjon er hva som gikk galt. Her gir den først *type* feil, og så er det en setning som prøver å forklare hva hva som skjedde. Vi skal nå med vilje lage et program med en feil, og så skal vi se på feilmelding vi får.

In [17]:

```
radius = 10
overflate = 4*pi*radius**2

print(overflte)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
 last)
<ipython-input-17-580945939324> in <module>()
      2 overflate = 4*pi*radius**2
      3
----> 4 print(overflte)

NameError: name 'overflte' is not defined
```

Feilmeldingen vi får er en `NameError` , det betyr at det er noe galt med *navnene* på variablene i programmet vårt. Vi sier at feilmeldingen sier at `overflte` ikke er definert, og det stemmer jo, fordi vi definerte `overflate` , og vi har en skrivefeil når vi prøver å printe ut. Her kan man kanskje tenke seg at Python burde skjønnt hva vi mente, men det gjør den altså ikke. Datamaskinen er egentlig ikke så veldig smart, den er bare veldig flink til å gjøre enkle kommandoer veldig fort.

Når du programmerer kommer det til å dukke opp mange `NameError`, enten fordi du har skrivefeil i variabelnavn, eller fordi du rett og slett glemmer å definere variabelen. Kanskje vi skriver opp volumberegningen for en kule, men så glemmer vi faktisk å definere radiusen, da vil programmet kræsje. Det høres kanskje litt dramatisk ut med begrepet *kræsje*, men heldigvis gjør det ingenting at et program, eller en celle i Jupyter, kræsjer. Det skader ikke maskinen, og gjør ingenting. Det eneste vi trenger å gjøre er å finne feilen, rette den opp, og så kjøre programmet eller cellen på nytt.

Det å gjøre feil, og så finne dem og rette dem opp er en enormt stor del av programmering, også for de som har programmert i mange år er det sånn. Det er ingenting galt med å få en feilmelding, det er en del av prosessen - og de verste feilene å finne er de som faktisk gjør at programmet gir feil resultat, istedenfor å kræsje! Tenk for eksempel at vi i volumberegningen vår skriver opphøyd i 2 istedenfor 3? Da blir jo svaret feil, men vi får ingen beskjed om det.

Når vi jobber med koding er det en god vane og alltid lese feilmeldingen når noe går galt. De er ofte litt kryptiske og vanskelig å forstå, men om man ihvertfall prøver å lese og forstå dem vil man begynne å gjenkjenne feil man har gjort før, og begynne å lære seg hva de betyr.

Eksempeloppgaver: Finn fem feil

En annen god type oppgave for å lære bort programmering er å lage små programmer som med vilje har en del feil innebygget - og så ber man elevene om å finne feilene. Her kan man jobbe med programmene på egenhånd, i par, eller hele klassen kan gjør det sammen. For å forstå hva som går galt kan man prøve å kjøre programmet og lese feilmeldingene, eller man kan prøve å etterligne datamaskinen og gå igjennom programmet for hånd, steg for steg.

Det kan være lurt å informere elevene om hvor mange feil programmet har. Fem kan i mange tilfeller være litt mange, og 2-4 er nok et fint antall for kortere programmer. Etterhvert som elevene finner feilene må de så rette dem opp. På denne måten må de tenke igennom hvordan det kan gjøres riktig, i tilegg til å forstå hvorfor det gikk galt. Når man har rettet om én feil, så kan man kjøre programmet på nytt og få mer informasjon

Det følgende programmet har 3 feil. Finn dem og rett dem opp.

In [ ]:

```
pi = 2.14
radius = 10

volum = 4*pi*radius**3
print(volm)
```

Her er første feil at vi har skrivefeil når vi skal bruke `print`, dette gir en `NameError`. De andre feilene gjør ikke at programmet kræsjer, men at vi får feil svar. Det første er at π er satt til 2.14 istedenfor 3.14, og at vi glemmer å dele på 3.

Oppgaver av denne typen er også utvidbare, og kan basere seg på nye konsepter og teknikker elevene har lært seg.

Feilmelding: SyntaxError

Vi har sett på vår første feilmelding, som var en `NameError`, og dette er en vanlig feil å få. La oss se på en annen veldig vanlig feil `SyntaxError`. Ordet *syntaks* i denne konteksten betyr språkreglene. I det norske språk har vi et sett regler som forteller oss hvordan vi skal bygge opp setninger, og dette er *syntaksen* til språket. En `SyntaxError` oppstår når vi skriver kodelinjer som Python rett og slett ikke forstår, fordi vi ikke følger reglene til språket. La oss se på et par eksempler:

In [19]:

```
1 x = 10
2 y = 10
3 z = x y
```

```
  File "<ipython-input-19-c2434f0513f7>", line 3
    z = x y
          ^
SyntaxError: invalid syntax
```

Her får vi en `SyntaxError`, fordi Python skjønner ikke hva vi mener når vi skriver `x y`. Dette kan for eksempel skje om vi har glemt å skrive en operator som `+`. Vi ser at feilmeldingen vi får ikke er så beskrivende, og dette er jo litt forståelig, Python skjønner jo ikke bæret av hva vi skriver, så klarer derfor ikke si ifra hva som er galt. En annen vanlig måte å få `SyntaxError` på er å bomme på antall paranteser

In [20]:

```
1 print((3+4)/2
```

```
  File "<ipython-input-20-3cbe7b0b5d1c>", line 1
    print((3+4)/2
                ^
SyntaxError: unexpected EOF while parsing
```

Her blir også Python forvirret, vi har et sett paranteser for utregningen vår, men `print` funksjonen trenger også paranteser og her har vi glemt den siste slutt parantesen. Feilmeldingen nå er `unexpected EOF while parsing`, som også er meget kryptisk. Her betyr EOF *end of file*, så Python prøver å fortelle oss at den lette etter noe med møtte plutselig opp på enden av fila. Det viktige med denne feilmeldingen er egentlig linjenummeret, så vi kan lete oss frem til linja som skaper problemet, og deretter begynner detektivarbeidet.

Datatyper

Vi har diskutert hvordan variabler har et navn og en *verdi*, på engelsk kaller vi disse for *name* og *value*. Det at vi kaller det *verdi* er kanskje litt misvisende, for mange er nok vant til å assosiere ordet "verdi" med tall. Men i dette tilfellet trenger ikke verdien til en variabel å være tall, det kan for eksempel være tekst, sannhetsverdier, eller mer kompliserte ting. I tilegg til navn og verdi har derfor variabler en *type*, som sier hva slags innehold variabelen har. Noen syns det er mer forklarende å kalle disse *datatyper*. I noen programmeringsspråk må man si hva slags type variabelen skal ha når man oppretter den, i Python skjer dette automatisk når vi oppretter den.

Vi kan sjekke typen til en variabel med funksjonen `type()`.

In [21]:

```
antall = 3
pi = 3.14
navn = "Georg"

print(type(antall))
print(type(pi))
print(type(navn))
```

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

Her ser vi at `antall = 3` har gitt en variabel av typen `int`. Her står *int* for *integer*, som er engelsk for *heltall*. Den andre variabelen, `pi`, har blitt til typen `float`, som står for *flyttall*. Flyttall er spesielle desimaltall vi bruker på datamaskinen. Vi kaller disse bevisst ikke for desimaltall, for en datamaskin klarer ikke representere desimaltall perfekt. For en elev i ungdomsskolen kan man forenkle og si at *flyttall* bare er et navn for desimaltall. Til slutt har vi `navn` som har typen `str`, som er kort for *streng*, som igjen kommer fra *tekststreng*. Dette er bare de enkleste datatypene i Python, og de kan fort bli mer kompliserte. Senere skal vi for eksempel se på lister og arrays, som er variabler som inneholder en lang rekke ulike verdier. Merk at alt som får et navn i Python er en form for variabel, dette inkluderer funksjoner. Ta for eksempel `print` funksjonen:

In [22]:

```
print(type(print))
```

```
<class 'builtin_function_or_method'>
```

Her ser vi at `print` egentlig bare er en variabel av typen "builtin_function". Her betyr *built in* bare at den er innebygd i Python, så vi ikke trenger å opprette den selv.

Når vi gjør operasjoner på variabler, så vil Python tolke disse operasjonene forskjellig avhengig av hva slags datatype variablene er, ta for eksempel addisjon av to variabler, som vi skriver som `a + b`. Her vil det som skjer avhenge helt av hva slags type variabler `a` og `b` er. Om de er tall er det ganske rett frem

In [23]:

```
print(3 + 4)
print(2.4 + 1.0)
```

```
7
3.4
```

Men hva skjer om `a` og `b` er tekststrenger for eksempel?

In [24]:

```
print("Hei" + "på" + "deg!")
print("3" + "4")
```

```
Heipådeg!
34
```

Vi ser at tekststrenger som adderes skjøtes sammen, også om det er tekst av tall. Her gjelder det altså å ha litt kontroll på at variablene våres har de typene vi tror de har, ellers kan det fort bli surr. La oss se på et eksempel hvor dette går galt: Vi ønsker å lage en enkel kalkulator, som tar to tall inn og legger dem sammen:

In [25]:

```
a = input("Gi et tall:      ")
b = input("...og så et til: ")
print("{} + {} = {}".format(a, b, a+b))
```

```
Gi et tall:      7
...og så et til: 8
7 + 8 = 78
```

Her blir jo svaret helt feil! Dette er fordi `input` vil tolke alt brukeren gir som tekststrenger. Og når vi adderer tekst skjøtes det. Her vi må altså først gjøre om teksten vi får fra brukeren til tall, og så addere. Dette kan vi gjøre ved å skrive `int(a)` om vi vil ha heltall, eller `float(a)` om vi vil ha desimaltall.

In [26]:

```
a = int(input("Gi et tall:      "))
b = int(input("...og så et til: "))
print("{} + {} = {}".format(a, b, a+b))
```

```
Gi et tall:      7
...og så et til: 8
7 + 8 = 15
```

Tilsvarende kan vi gå fra tall til tekst ved å bruke `str(a)` .

Lister

La oss si vi ønsker å se på temperatursvingninger iløpet av dagen. Vi har derfor notert hva termometeret vårt viser på klokkeslettene 9:00, 12:00, 15:00, 18:00 og 21:00. Vi er nå klare for å legge inn resultatene i en notebook for å analysere dem litt. Her kunne vi lagret en variabel for hver måling, men dette er litt mye jobb. Det skalerer også veldig dårlig, for si vi hadde satt opp en automatisk måling hvert minutt, da hadde vi fått nesten tusen målepunkter og det blir uholdbart å skulle opprette så mange variabler hver for seg.

Vi introduserer derfor en datatype vi ikke har sett på enda: *lister*. Lister er en samling av andre objekter i Python. Hvis vi tenker på en variabel som en boks med et bestemt innhold kan vi tenke på en liste som en boks med mange individuelle rom. Hver rom har sitt eget innhold. Vi kan opprette en liste for målepunktene våres ved å skrive dem innenfor firkantparanteres `[]` og dele de forskjellige målepunktene med komma:

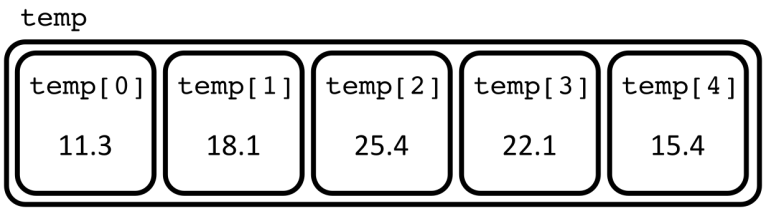
In [27]:

```
temp = [11.3, 18.1, 25.4, 22.1, 15.0]
print(type(temp))
```

```
<class 'list'>
```

Nå vil `temp` være en liste av tall. Hvert målepunkt, eller tall, er et *element* i lista. Vi kan refere til alle målepunktene som ett ved å referere til `temp`, men vi kan også referere til bestemte elementer i lista ved å skrive for eksempel `temp[2]`, der tallet forteller hvilket element det er snakk om. Det som er litt lite intuitivt her, er at Python *begynner å telle fra 0*. Så `temp[2]` er ikke det andre elementet i lista, men det tredje. Dette høres kanskje veldig rart ut, men det er ganske vanlig i programmeringsverden å telle fra 0, og det finnes det faktisk gode grunner til, men la oss ikke gå inn på det.

Vi kan tegne opp lista med målepunkter som følger



In [29]:

```
print(temp)
print(temp[2])
```

```
[11.3, 18.1, 25.4, 22.1, 15.0]
25.4
```

Lister er veldig viktige i programmering, for de larg oss behandle samlinger av informasjon. Lister trenger ikke bestå av tall, men kan være hva som helst, for eksempel kan vi ha en liste med navn

In [30]:

```
gruppe_a = ["Andreas", "Beate", "Christine", "Daniel"]
```

Det går også fint å *blande* forskjellige datatyper i en liste. La oss for eksempel si vi har en adressebok, der vi for hver person i adresseboken skal ha et navn, en e-postadresse og et telefonnummer, det kan vi gjøre som følger

In [31]:

```
ola = ["Ola Nordmann", 80045300, "ola@norge.no"]
```

Når en liste er blitt laget, så er ikke størrelsen på lista satt i stein, vi kan utvide lista ved å legge til elementer, eller vi kan korte den ned ved å fjerne elementer. La oss for eksempel si vi vil legge til flere målepunkter i temperaturlista vår, det kan vi gjøre med `append()` funksjonen. Her er funksjonen `append()` en litt spesiell funksjon, for den opererer på en bestemt liste, så vi kaller den som følger: `temp.append(8.6)`. En slik funksjon kalles for *en metode*, og dette er en *listemetode*. Append legger til det nye elementet på enden av lista som et nytt element. Vi kan sjekke lengden på en liste med `len`, som er kort for "length".

In [32]:

```
temp.append(8.6)
print(temp)
print(len(temp))
```

```
[11.3, 18.1, 25.4, 22.1, 15.0, 8.6]
6
```

Vi kommer snart til å se på tilfeller der vi bygger opp en liste fra bunn av, element for element. Da må vi først opprette en tom liste, det gjør vi ved å skrive tomme firkantparanteser: `tom_list = []` . Deretter bruker vi `append` for å legge til element for element. Vi kan også skjøte sammen lister med `+` .

Løkker

Når vi ønsker å gjenta en prosess mange ganger bruker vi en *løkke*. Løkker er et av de viktigste konseptene i programmering. Python har to typer løkker: *for*-løkker og *while*-løkker. Vi skal se på begge to, men vi begynner med *for*-løkka. En *for*-løkke jobber seg igjennom en liste (eller lignende datatyper som består av flere elementer), og gjør noe med hvert element i lista.

La oss si vi ønsker å jobbe med temperaturmålepunktene vi lagde tidligere. Da lager vi en løkke som følger

```
for målepunkt in temp:
    <kode som gjentas for hvert målepunkt>
    <kode som gjentas for hvert målepunkt>
    <kode som gjentas for hvert målepunkt>

<første kodelinje som utføres når løkka er ferdig>
```

Denne koden vil gå igjennom temperaturlista vår, og gjenta en bolk med kode for hvert element i lista. Her gir vi navnet `målepunkt` til en variabel som settes til det elementer vi *løkker over*, denne variabelen vil altså endre seg for hver gang kodeblokken utføres. Ordene `for` og `in` må vi ha med, og `temp` er bare navnet på lista vår. Til slutt bruker vi et kolon. Etter dette har vi et sett med kodelinjer (dette kan være en enkelt linje, eller en lang rekke linjer), og det er disse kodelinjene som vil gjentas. Merk at hver kodelinje som skal gjentas har fått et innrykk på venstre side, dette innrykket kan man lage med `Tab` knappen, som du finner over Caps Lock, eller man kan bare skrive en del mellomrom på rad. Alle kodelinjer med samme innrykk hører til løkka, men så fort vi har en linje uten innrykk er løkka ferdig.

La oss se på et enkelt eksempel

In [33]:

```
for målepunkt in temp:
    print(målepunkt)
```

```
11.3
18.1
25.4
22.1
15.0
8.6
```

I dette eksempelet blir kodelinja `print(målepunkt)` gjenntatt seks ganger på rad, men for hver gang blir først `målepunkt` variabelen satt til å være det neste elementet i lista. La oss si vi ønsker å regne ut gjennomsnittstemperaturen, dette kan vi gjøre ved å først summere alle temperaturene i lista, og så dele på antall målepunkter. Det kan vi gjøre med en løkke som følger

In [34]:

```
total = 0
for målepunkt in temp:
    total += målepunkt

gjennomsnittstemperatur = total/len(temp)
print(gjennomsnittstemperatur)
```

```
16.75
```

Her lager vi først variabel som skal holde styr på totalen. Så løkker vi over lista og legger til hver temperatur til totalen. Merk at vi ikke bør kalle tellevariabelen vår for `sum` , da det er en egen funksjon i Python. Etter løkka har vi funnet summen av alle målepunktene, så vi kan da regne ut gjennomsnitstemperaturen ved å dele totalen på antall målepunkter, som vi finner ved å bruke `len()` .

Eksempel: Renteberegninger

La oss si vi ønsker å vite hvor lønnsomt det er å penger på sparekonto. La oss si banken vår gir oss en effektiv årlig rente på 3.45% på sparekontoen vår. La oss si vi setter inn 10000 kroner. Hvordan vokser pengene på kontoen over de neste 10 årene? For å finne ut dette kan vi bruke en løkke

In [35]:

```
penger = 10000

for år in [1, 2, 3, 4, 5]:
    penger *= 1.0345
    print("Etter {} år er det {} kr på kontoen".format(år, penger))
```

```
Etter 1 år er det 10345.0 kr på kontoen
Etter 2 år er det 10701.9025 kr på kontoen
Etter 3 år er det 11071.11813625 kr på kontoen
Etter 4 år er det 11453.071711950624 kr på kontoen
Etter 5 år er det 11848.20268601292 kr på kontoen
```

Her definerer vi først en variabel `penger` til å være 10000, siden det er startsummen vår. Deretter løkker vi over 1 til 10 år, for hvert år ganger vi med `1.0345` for å finne det som står på konto etter et nytt år har gått. Til slutt skriver vi ut resultatet. Vi ser dette fungerer bra, men utskriften blir ikke så veldig pen, ettersom at vi får fryktelig mange desimaltall på summen vår. Antall øre er ikke så spennende, så vi fikser dette ved å spesifisere at vi skal skrive ut med 0 desimaler, da runder Python til nærmeste krone når det skrives ut

In [36]:

```
penger = 10000

for år in [1, 2, 3, 4, 5]:
    penger *= 1.0345
    print("Etter {} år er det {:.0f} kr på kontoen".format(år, penger))
```

```
Etter 1 år er det 10345 kr på kontoen
Etter 2 år er det 10702 kr på kontoen
Etter 3 år er det 11071 kr på kontoen
Etter 4 år er det 11453 kr på kontoen
Etter 5 år er det 11848 kr på kontoen
```

Så etter 5 år har vi en fortjeneste på nesten 2000 kroner, ikke verst! Det er mange eksempler av denne typen hvor vi ønsker å løkke over tallrekker. Si at vi for eksempel ønsker å gjøre det samme, men for 30 år. Det blir fort kjedlig om vi trenger å skrive ut en liste med alle tallene fra 1 til 30. Her finnes det en funksjon som gjør jobben for oss, den heter `range()` og gir oss en tallrekke, la oss se hvordan den virker.

`range` funksjonen

Funksjonen `range` kan brukes om vi ønsker å løkke over en tallrekke, da kan vi skrive

```
for tall in range(a, b)
    <kodeblokk>
```

Og vi løkker da over alle heltall fra og med `a`, til (men ikke med), `b`.

In [37]:

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

```
1
2
3
4
```

Vi kan slenge på et tredje tall som input til `range`, og det gjør vi hvis vi ikke ønsker å gå i steg på 1 og 1. La oss si vi ønsker å skrive ut 5-gangen for eksempel, da kan vi gjør det slik:

In [38]:

```
for tall in range(0, 30, 5):
    print(tall)
```

```
0
5
10
15
20
25
```

Eller om vi ønsker å skrive ut litt mer fancy:

In [39]:

```
for tall in range(0, 30, 5):
    print("{:.0f} x 5 = {}".format(tall/5, tall))
```

```
0 x 5 = 0
1 x 5 = 5
2 x 5 = 10
3 x 5 = 15
4 x 5 = 20
5 x 5 = 25
```

Vi kan også gi bare ett tall til `range`, isåfall starter den på 0, og går opp til, men ikke med, tallet vi sender inn:

In [40]:

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

```
0
1
2
3
4
```

Dette betyr at dersom vi ønsker å gjenta en kodesnutt n ganger, så kan vi gjør dette ved å skrive

```
for tall in range(5):
    <kode>
```

Merk at vi kan gjøre dette selv om vi ikke bruker tallene til noe. Ta for eksempel adjektivhistorien vår tidligere, si vi vil ha 5 adjektiv fra brukeren, da kan vi skrive:

In [41]:

```
adjektiv = []
for tall in range(5):
    adjektiv.append(input("Gi et adjektiv: "))

print(adjektiv)
```

```
Gi et adjektiv: blå
Gi et adjektiv: søt
Gi et adjektiv: skjør
Gi et adjektiv: sint
Gi et adjektiv: teit
['blå', 'søt', 'skjør', 'sint', 'teit']
```

Om vi nå skal skrive ut historien kan vi bruke adjektiv lista ved å skrive `.format(*adjektiv)`, her gjør `*` at listen tolkes som en rekke enkelete variabler.

Merk at vi i dette eksempelet ikke faktisk bruker *tallet vi løkker over* til noe - og det gjør ingenting - målet vårt er bare å slippe å skrive den samme koden gang på gang. Da kan man eventuelt vurdere å skrive

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

For å gjøre det tydelig at her gjentar vi bare noe 5 ganger, det er ikke slik at vi egentlig prøver å løkke over noen tall å gjøre ting med disse.

Vanlig feil: Glemt kolon

En veldig vanlig feil blant nybegynnere er å glemme kolonet på slutten av begynnelsen av for-løkka, la oss se hva slags feilmelding det gir

In [42]:

```
for tall in range(10)
    print(tall)
```

```
  File "<ipython-input-42-9542321e44ec>", line 1
    for tall in range(10)
                         ^
SyntaxError: invalid syntax
```

Igjen får vi en lite beskrivende `SyntaxError` , men vi ser at den peker på akkurat der kolonet skulle stått.

Å løkke over andre ting en lister

Det er mulig å løkke over andre ting en lister, alt som består av et sett med elementer kan løkkes over. Det går også ann å løkke over tekststrenger for eksempel, da tolkes hvert tegn i strengen som et eget element

In [43]:

```
setning = "Hello, World!"
for tegn in setning:
    print(tegn)
```

```
H
e
l
l
o
,

W
o
r
l
d
!
```

Dette kan vi bruke til å for eksempel regne ut *tverrsummen* til et tall. Tverrsummen er summen av alle siffrene i tallet hver for seg. Vi kan hente ut disse ved å først endre tallet til en tekststreng, så gå igjennom siffer for siffer med en løkke, og for hvert siffer kan vi gjør om tilbake til et tall og legge sammen

In [44]:

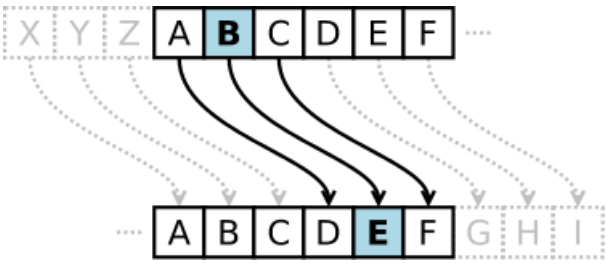
```
tverrsum = 0
tall = 481952
for siffer in str(tall):
    tverrsum += int(siffer)

print(tverrsum)
```

29

Tverrsummen kan blant annet brukes til å se hva et tall er delelig med (https://no.wikipedia.org/wiki/Tverrsum), her ser vi enkelt at det originale tallet (481952) ikke er delelig med 3.

Eksempelprosjekt: Enkrypsjon og hemmelige beskjeder

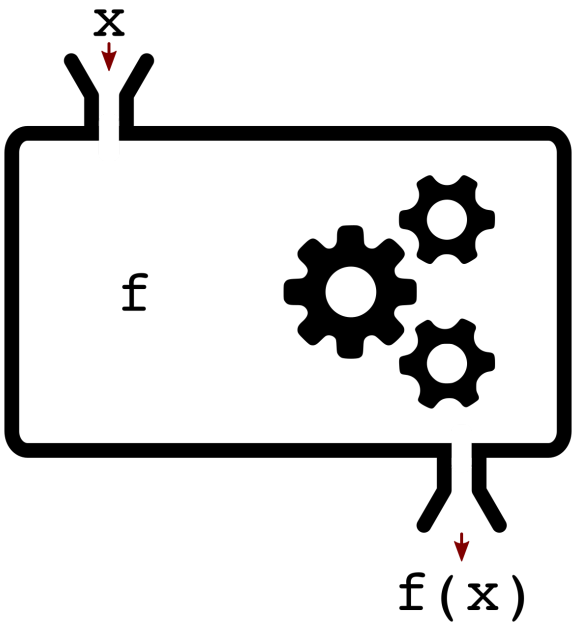
En annen stilig ting vi kan gjøre når vi løkker over en tekst er å lage hemmelige beskjeder. Dette kan vi for eksempel gjøre med et *Cæsarchiffer* (https://no.wikipedia.org/wiki/C%C3%A6sarchiffer) der hver bokstav i en beskjed flyttes et gitt antall hakk fremover eller bakover i alfabetet.



Om du ønsker å lage et prosjekt eller opplegg ut av dette har Kidsa koder et bra oppgavesett (http://oppgaver.kidsakoder.no/python/hemmelige_koder/hemmelige_koder.html) på dette temaet som du kan bruke som inspirasjon og hjelp.

Funksjoner

Vi har allerede sett eksempler på et par funksjoner, for eksempel `print`, `input`, og `sqrt` er alle funksjoner. En funksjon er egentlig som alle andre variabler i Python, de har et navn og et innhold. I dette tilfellet er innholdet en egen kode som kjøres når vi *kaller på funksjonen*. Når vi kaller på funksjonen kan vi sende med en input, og vi får en output tilbake. Vi kan tenke på en funksjon som en slags maskin, der vi mater inputten in på den ene siden, så får vi et resultat ut på den andre siden. Når vi kaller på en funksjon skriver vi `f(x)`, der `f` er navnet på funksjonen, og `x` er inputen.



Med denne analogien ser vi også at vi ikke nødvendigvis trenger å vite noe om hva som skjer inni funksjonen for å bruke den, det vi er interessert i er sluttresultatet. På denne måten fungerer funksjoner som abstraksjoner av idéer og konsepter, der vi kan gjemme de stygge detaljene på insiden av boksen.

I tilegg til å kunne skjule detaljer, er funksjoner fine for å unngå å gjenta kode. La oss ta volumberegningen vi brukte tidligere, om vi ønsker å regne ut volumet av kuler av forskjellig radius må vi egentlig gjenta beregninger vår mange ganger. Eller vi kan lage en funksjon som gjør det én gang, og så kalle den mange ganger.

In [45]:

```
1 def volum(radius):
2     return 4*pi*radius**3/3
```

Her ser vi at syntaksen for å lage en funksjon er

```
def navn_på_funksjon(input):
    <kode>
    <kode>
    return output
```

Her står `def` for *define*, fordi vi definerer en funksjon. Akkurat som for løkkene vi så på tidligere bruker vi kolon på slutten av første linje, og deretter innrykk på alle linjene som skal høre til funksjonen.

Når vi definerer en funksjon skjer det ikke stort, det vi gjør er å opprette en ny variabel, i vårt eksempel, `volum` , som er en funksjon.

In [46]:

```
1 print(type(volum))
```

```
<class 'function'>
```

Koden inne i volumfunksjonen vil ikke kjøres før vi bruker funksjonen ved å kalle på den:

In [47]:

```
print(volum(1))
print(volum(2))
print(volum(3))
```

```
4.1866666666666665
33.49333333333333
113.04
```

Det som skjer når vi kaller på funksjonen er at koden inne i funksjonen kjøres. Først settes inputvariabelen `radius` inne i funksjonen til det vi sender inn, så regner den ut volumet og bruker `return` for å sende volumet tilbake til brukeren. Her skriver vi det ut, men vi kunne for eksempel lagret det til en variabel

In [48]:

```
fotball_radius = 11
fotball_volum = volum(11)
```

Som alltid tolker Python det som er på høyre side av `=` først, dette betyr at funksjonen kalles, og vi får et resultat tilbake som er et gitt tall, deretter gir vi denne verdien til variabelen på venstre side.

Input og Output

I vårt eksempel har vi én input, og´en output, men dette kan variere. En funksjon kan også ha *ingen* inputs og/eller *ingen* outputs, eller flere enn én. Om vi ønsker fler inputs skriver vi dem bare etterhverandre, separert med komma:

In [49]:

```
def legg_sammen(a, b):
    return a + b
```

Tilsvarende kan vi gi flere svar tiblake om vi lister dem opp i rekkefølge, separert med komma

In [50]:

```
def kule(radius):
    omkrets = 2*pi*radius
    overflate = 4*pi*radius**2
    volum = 4*pi*radius**3
    return omkrets, overflate, volum
```

In [51]:

```
fotball_radius = 11
fotball = kule(fotball_radius)
print(fotball)
```

```
(69.08, 1519.76, 16717.36)
```

En funksjon trenger ikke å ta noen input, da lar vi bare parantesene stå blanke, både når vi definerer og kjører funksjonen.

In [52]:

```
def velkommen():
    print("Hei! Velkommen til Python!")

velkommen()
```

```
Hei! Velkommen til Python!
```

Tilsvarende ser vi at en funksjon ikke trenger å ha en `return` kodelinje. Funksjonen vi nettopp skrev printer noe til skjermen, og returnerer ingenting. Noe litt interessant skjer med funksjoner som ikke returnerer noe, la oss se på et eksempel

In [53]:

```
beskjed = velkommen()
```

```
Hei! Velkommen til Python!
```

Etter å ha kjørt denne kodelinja kan man kanskje tro at `beskjed` variabelen inneholder tekststrengen "Hei! Velkommen til Python!", men det gjør den faktisk ikke! Husk at når vi oppretter variabler vil Python alltid tolke høyre siden først. Her kjører `velkommen` funksjonen, som printer noe til skjerm, men *resultatet* fra kjøringen er det funksjonen returnerer, og funksjonen vår returnerer jo ingenting. Samtidig får vi ingen feilmelding, så hva er det som lagres i `beskjed` da? La oss se:

In [54]:

```
print(beskjed)
```

```
None
```

I Python returnerer alle funksjoner noe, selv de som ikke har en `return` kodelinje, og det de returnerer er `None`. Dette er vanlig i mange programmeringsspråk. Dette er også en vanlig feil å få, enten fordi man har glemt å bruke `return` i en funksjon man lager selv, eller fordi man blingser på hvordan en funksjon faktisk virker. For eksempel vil kodelinjen

```
kvadratrot = print(sqrt(9))
```

kjøre uten å gi feilmelding, men den gjør neppe det man hadde tenkt. For her tror man kanskje svaret skrives ut og lagres til variabelen, men det gjør den altså ikke. Isteden må man dele opp i to linjer

In [55]:

```
kvadratrot = print(sqrt(9))
print(kvadratrot)
```

```
3.0
None
```

In [56]:

```
kvadratrot = sqrt(9)
print(kvadratrot)
```

```
3.0
```

Eksempeloppgave: Konvertering av temperatur

Man kan oppgi temperaturer i forskjellige målestokker, de to vanligste målestokkene er Celsius og Fahrenheit. I Norge bruker vi Celsius, så om man får oppgitt en temperatur i Fahrenheit er det vanskelig å skjønne hva det betyr. Man kan regne om fra Fahrenheit til Celsius fra formelen

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

Oppgave a) Lag en funksjon som heter `F2C` som tar en temperatur i Fahrenheit inn, og gir den tilsvarende temperaturen i celsius tilbake.

Oppgave b) En kakeoppskrift sier at kaken skal bakes på 350 grader Fahrenheit. Bruk funksjonen du skrev til å sjekke hvor mange celsius det er. Virker svaret ditt rimelig?

Oppgave c) Lag den motsatte funksjonen, `C2F`, altså en funksjon der du sender antall grader Celsius inn og får antall grader Fahrenheit tilbake. Her må du først endre på det matematiske uttrykket.

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

Eksempeloppgaver: Løkker for Hånd

Her kommer vi igjen tilbake oppgaver der man må "leke datamaskin" og gå igjennom for hånd. Dette er en god måte å få elevene til å tenke igjennom hva løkkene egentlig gjør.

For hver kodesnutt, forutsi hva som skrives ut. Kjør kodene og sjekk om du har rett

Oppgave a)

```
for i in [0, 2, 4]:
    print(2*i - 1)
```

Oppgave b)

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

Oppgave c)

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

Oppgave d)

```
for i in [-7, 0, 7]:
    print(i**2 - i)
```