# Løse reelle problemer

## Litt mer om løkker, prosedyrer, funksjoner, tekst og innlesing fra fil

INF1000, uke4
Geir Kjetil Sandve

# Tilbakeblikk

- Dere bør nå beherske det sentrale fra uke 1 og 2:
  - Uttrykk, typer, variabler, beslutninger (if)
- Forrige uke introduserte to helt fundamentale konsept
  - **while** for å kjøre en kodeblokk mange ganger
  - **liste** for å holde på mange verdier
- Hvor mye må dere ha skjønt av lister og løkker?
  - Ikke forventet stålkontroll i dag - krever modning
  - Det løsner etterhvert - Mengde, mengde, mengde
  - Mindre enn 13 timer med faget er på eget ansvar

# Outline

- For-løkker
- Prosedyrer med parametre
- Funksjoner
- Lese fra fil
- Mer om lister og strenger
- Jobbe med ekte data

# Outline

- For-løkker
- Prosedyrer med parametre
- Funksjoner
- Lese fra fil
- Mer om lister og strenger
- Jobbe med ekte data

# Iterere gjennom en samling: **for**

- Syntaks:
  - ```
    for variable in container:
        statement1
        ...
    ```
- Eksempel:
  - ```
    for tall in [1,2,3,4]:
        print(tall*tall)
    ```
- En løkke som kjøres én gang med hver verdi i en samling (container)
  - Variabelen mellom "for" og "in" blir satt til én verdi fra samlingen for hver gang kodeblokken i løkka kjøres
- [sum_vha_for_v1.py]

# En samling (container)

- En samling er en verdi som består av flere elementer
  - En liste, en mengde, ...
    - Alt som kan itereres gjennom med en for-løkke
- Noen samlinger kan også aksesseres på indeks:
  - Mulig for liste: min_liste[2]
  - Ikke mulig for mengde: ~~min_mengde[2]~~

# Strenger er også en type lister!

- Man kan iterere gjennom strenger:
  - for bokstav in "NORGE":
    print("Gi meg en " + bokstav)
- Man kan aksessere strenger på indeks:
  - land = "NORGE"
    print( land[2])
- Men man kan ikke endre en streng:
  - ~~land[2] = "J"~~
  - Man sier at strenger er *uforanderlige* (immutable)

# Kjapt spesifere lister av tall: *range*

- Funksjonen range kan brukes for å kjapt lage lister:
  - assert range(5) == [0,1,2,3,4]
  - assert range(2,5) == [2,3,4]
  - assert range(0,5,2) == [0,2,4]
  - assert range(1,5,2) == [1,3]
- Dette kan brukes som del av for-løkke
  - ```
    for tall in range(1,5):
        print(tall*tall)
    ```
- [sum_vha_for_v2.py]
    ```

# Outline

- For-løkker
- Prosedyrer med parametre
- Funksjoner
- Lese fra fil
- Mer om lister og strenger
- Jobbe med ekte data

# Flere typer subrutiner

- **Subrutine** (fra uke 2): en **navngitt blokk** med kodelinjer, som kan **kalles** og **tilpasses**
- Vi vil i dag introdusere flere aspekter
  - Uke 2: Prosedyre - **uten** parametre og returverdi
  - I dag: Prosedyre - med **parametre**
  - I dag: Funksjon - med **returverdi**
  - Neste uke: **Instans**-metode (OO)

# Prosedyrer **med** parametre

- Prosedyren vi så på i tidligere uke gjorde alltid eksakt det samme når den ble kallet
  - Det er sjelden av nytte!
- For å være nytig må en slik prosedyre kunne **tilpasses**
  - Det gjør vi ved å sende inn **parametre**

# Din første prosedyre med parametre

- **`print`** er en prosedyre hvor utfallet tilpasses!
- `print(text)`:
  skriver verdien i `text` til skjermen†
  - Variabelen text er en **parameter**
  - Verdien vi gir inn (`hallo INF1001`) når vi kaller `print` er et **argument**
- Parameter og argument er to sider av det samme
  - Parameter: **variabel** i prosedyre som tar i mot verdi
  - Argument: **verdi** sendt inn når prosedyren kalles

# Prosedyre med parametre

```
def mittProsedyreNavn(parameter1, parameter2, ...):
  kodelinje1
  kodelinje2
  ...
```

For å kjøre alle kodelinjene i prosedyren ("*kalle* prosedyren"):

```
mittProsedyreNavn(argument1, argument2, ...)
```

# Prosedyrer outsourcer detaljer og håndterer redundans

- Man har ofte behov for (omtrent) samme funksjonalitet ulike steder i en kode
- Man ønsker da ikke å duplisere koden
  - Minsker oversiktlighet av kode
  - Endringer og rettinger må utføres mange steder
- Man samler i stedet funksjonaliteten i en prosedyren og kaller metoden der den trengs
  - Dersom det er noe variasjon i hva man trenger, representerer man det som varierer med en parameter

# Prosedyre med parametre

- {prosedyre_med_parameter_v1.py-prosedyre_med_parameter_v3.py}

# Outline

- For-løkker
- Prosedyrer med parametre
- Funksjoner
- Lese fra fil
- Mer om lister og strenger
- Jobbe med ekte data

# Subrutiner med returverdi: **Funksjoner**

```
def mittFunksjonsNavn(parameter1, ...):
    kodelinje1
    kodelinje2
    return beregnet_verdi
```

For å kjøre alle kodelinjene i funksjonen ("*kalle* prosedyren"):

```
verdien_jeg_trenger = mittFunksjonsNavn(argument1, ..)
```

# Subrutiner med returverdi: **Funksjoner**

- En funksjon lar deg outsource en beregning til en separat kodeblokk
  - Funksjonen tar inn en eller flere verdier,
    gjør en bestemt beregning,
    og sender tilbake resultatet
- Det som skiller en funksjon fra en prosedyre er altså at en funksjon returnerer noe:
  - ```
    def min_funksjon(parametre, ..):
        ...
        return en_beregnet_verdi
    ```
    ```

# Subrutiner med returverdi: **Funksjoner**

- Eksempler på funksjoner:
  - def gang_med_to(tall):
    return tall*2
  - def lag_velkomst(fag, person):
    return "Velkommen til " + fag + " kjære " + person
- Bruk av funksjoner

```
dusin=13
toDusin = gang_med_to(dusin)
print(toDusin) #Skriver ut 26

velkomst = lag_velkomst("inf1001", "Geir")
print(velkomst) #Velkommen til inf1001 kjære Geir
```

# Funksjoner kan enkelt testes

- Man vet hva funksjonen skal gjøre, og kan skrive testen før selve funksjonen!
  - `assert gang_med_to(3) == 6`
- Man kan ikke teste alle muligheter, men forsøk å dekke litt variert type argumenter
  *(ulikt som kunne tenkes å feile)*
  - ```
    assert gang_med_to(0) == 0
    assert gang_med_to(-3) == -6
    assert gang_med_to(2.4) == 4.8
    ```
    ```

# Funksjoner kan enkelt testes

- Skriv deretter gjerne en (tom) funksjon som skal feile
  - ```
    def gang_med_to(tall):
        return 0
    ```
- Forsøk til slutt å skrive riktig funksjon
  - ```
    def gang_med_to(tall):
        return tall*2
    ```
  - Se at programmet (asserts) ikke lenger feiler ved kjøring
- [ganging.py]

# Prosedyrer versus funksjoner i Python

- Det er i Python ikke noe teknisk skille mellom prosedyre og funksjon
  - Begge kalles i Python "funksjoner" og har samme form:

```
def min_funksjon(parameter):
    ...
```

- Forskjellen er i vårt hode (vårt formål):
  - Det som skiller en "ekte" funksjon fra en prosedyre er at funksjoner returnerer en verdi man er interessert i
- Dette ser man som en **return** i koden, men også ved at resultatet av kallet brukes/tas vare på:
  - `toDusin = gang_med_to(dusin)`

# Utsett til i morgen, det du ikke trenger gjøre i dag

- Prosedyrer med returverdi tillater å utsette problemer!
  - Fokuser først på hva som trengs overordnet
  - Deretter gå løs på detaljene
- Eksempel:

```
kvm=60
postnr=0316
inntekt=503800
pris = regn_bolig_pris(kvm, postnr)
maks_laan = regn_kreditt_verdighet(inntekt)
if (maks_laan > pris):
  print("Yes!")
# Deretter skriv selve funksjonene..
```

# Med/uten parametre/returverdi

- (En litt filosofisk distinksjon...)
- **Prosedyrer uten** parametre:
  - Handler stort sett om kontrollflyt
- **Prosedyrer med** parametre:
  - Handler stort sett om tilpasset gjenbruk av kode
- **Funksjoner**:
  - Outsourcer en overordnet beregning (transformerer inn til ut)

# Tre kilder til funksjoner

- Innebygde funksjoner
  - print, int, input ...
  - Følger med Python og er alltid tilgjengelige
- Funksjoner i standard-biblioteket (ikke innebygde)
  - sqrt, ctime
  - Er del av en modul: math, time, ...
  - ```
    from time import ctime
    print(ctime())
    ```
- Egendefinerte funksjoner
  - def min_funksjon(parameter): ...
    ```

# Prøv selv (5 min)
## (Lett modifisert fra eksamen 2014)

Skriv en funksjon

```
def pris(gratis, alder):
```

Dersom parameteren **gratis** har verdien **True**, skal funksjonen *alltid* returnere 0. Dersom parameteren **gratis** har verdien **False** og verdien av **alder** er mindre enn 18, skal funksjonen returnere 100, ellers 200. Altså skal f.eks. kallet **pris (true, 10)** returnere 0, kallet **pris(false,10)** returnere 100 og kallet **pris(false, 50)** returnere 200.

# En mulig løsning

```
def pris(gratis, alder):
    if gratis:
        svar = 0
    elif alder < 18:
        svar = 100
    else:
        svar = 200

    return svar
```

# En annen mulig løsning
## *(returnere inni **if**)*

```
def pris(gratis, alder):
    if gratis:
        return 0
    elif alder < 18:
        return 100
    else:
        return 200
```

# En tredje mulig løsning
## *(bare rene **if** - ingen else..)*

```
def pris(gratis, alder):
    if gratis:
        svar = 0
    if (!gratis and alder < 18):
        svar = 100
    if (!gratis and alder >= 18):
        svar = 200

    return svar
```

# Evaluering av uttrykk som inneholder funksjonkall

- Vi har i INF1001 ofte skrevet følgende:
  - innlest = input("Skriv et tall")
  - tall = int(innlest)
- Dette er nøyaktig det samme som følgende:
  - tall = int(input("Skriv et tall"))
- I det ene tilfellet tas teksten vare på i en variabel (innlest) før den konverteres, i den andre brukes den direkte

# Evaluering av uttrykk som inneholder funksjonkall

- Hva som nøyaktig skjer:
  - `tall = int(input("Skriv et tall"))`

```
                input("Skriv et tall")  -> "55"
          int(        "55"                  ) -> 55
tall =          55
```

- input er en funksjon som her får en beskjed som argument, og evaluerer til teksten brukeren skriver
- int er en funksjon† som her tar verdien som input evaluerer til som argument, transformerer denne,
og evaluerer til det tilsvarende heltallet

# En liten nøtt

- Hva skjer egentlig her?
  - noe = input(input("Spørsmål: "))
    print("trtrtrtrtrt: " + noe)

# Outline

- For-løkker
- Prosedyrer med parametre
- Funksjoner
- Lese fra fil
- Mer om lister og strenger
- Jobbe med ekte data

# Innlesing fra fil

- Å hente data fra filer er gøy!
- Man kan jobbe på mye større og mer spennende data enn fra tastatur
- Man slipper å taste det inn hver gang man kjører
- Innlesing fra enkle filer er veldig rett frem i Python

# Innlesing fra fil

- Å hente data fra filer er gøy!
  - Man kan jobbe på mye større og mer spennende data enn fra tastatur
  - Man slipper å taste det inn hver gang man kjører
- Innlesing fra tekstfiler er veldig rett frem i Python
  - En tekstfil åpnet i Python er faktisk en samling av linjer
  - Man kan dermed iterere gjennom linjer i filen vha for-løkke
  - (Det finnes også mange andre måter å gjøre det på)

# Hvordan lese inn fra tekstfil

- Først åpne en fil:
  - `min_fil = open("mittFilNavn.txt")`
- Deretter iterere gjennom hver linje i filen:
  - ```
    min_fil = open("mittFilNavn.txt")
    for linje in min_fil:
    ```
- Inni for-løkken kan man gjøre noe med linjen
  - ```
    min_fil = open("mittFilNavn.txt")
    for linje in min_fil:
        print("Her fant jeg: " + linje)
    ```
- [les_fra_fil.py]

# Alternative måter å lese fra fil på

- Kan lese én linje
  - **`linje = min_fil.readline()`**
  - returnerer tom streng ("") når den har nådd slutten av filen
- Kan lese hele filen som en liste av linjer (liste av streng-verdier)
  - **`alle_linjer = min_fil.readlines()`**
- Kan lese hele filen som én streng-verdi (inkludert linjeskift)
  - **`hele_teksten = min_fil.read()`**

# Hvordan skrive til tekstfil

- Først åpne en fil for skriving:
  - `min_fil = open("mittUtFilNavn.txt", "w")`
- Deretter skrive tekst (en streng-verdi) til filen:
  - `min_fil = open("mittUtFilNavn.txt", "w")`
    **`min_fil.write("Dette havner i filen")`**
- Man vil typisk ha linjeskift i det man skriver ut:
  - **`min_fil.write("Dette havner i filen\n")`**
    **`min_fil.write("Dette havner på\n ulike\n linjer\n")`**
- Når man er ferdig bør man lukke filen:
  - `min_fil.close()`
- [elefanter.py]

# Outline

- For-løkker
- Prosedyrer med parametre
- Funksjoner
- Lese fra fil
- Mer om lister og strenger
- Jobbe med ekte data

# Mer om lister

- Kakebit (slice)
  - liste = [1945, 1814, 1905, 1945]
  - assert liste[1:3] == [1814, 1905]
- Sortere
  - assert sorted(liste) == [1814, 1905, 1945, 1945]
- Telle
  - assert liste.count(1945) == 2
- Summere
  - assert sum(liste) == 7609

# Mer om strenger:
## *Siden strenger er lister*

- tekst = "kamel"
- assert tekst[0:3] == "kam"
- assert sorted(tekst) == "aeklm"
- assert tekst.count("m") == 1
- assert sum(tekst)==1000 #(summere gir dog ikke mening)

# Mer om strenger

- **Strip** fjerner blanke i endene av strengen (kun endene)
  - setning = "  egg,hvete,epler og kanel  "
  - assert setning.strip() == "egg,hvete,epler og kanel"
  - setning = setning.strip()
- **Split** deler opp strengen i flere deler (liste av strenger)
  - assert setning.split(",") == ["egg", "hvete", "epler og kanel"]
  - ordene = setning.split(",")
  - assert len(ordene) == 3
  - assert ordene[2] == "epler og kanel"

# Mer om utskrift

- Vi har til nå konkatenert tekst ifbm utskrift
  - print( "Hei" + "INF" + "1000")
  - Altså konkateneres 3 strenger til én streng-verdi som er argument til funksjonen print
- Biblioteksfunksjonen *print* kan ta imot mange argumenter! §
  - print("Hei", "INF", "1000")
- Argumentene trenger ikke være strenger (blir konvertert automatisk)
  - print("Hei INF", 500*2, "er", 1==1)

§*(hvordan kan det være mulig? - kommer senere)*

# Oppsummering

- Prosedyrer og funksjoner gjør livet behagelig!
  - Tillater å tenke overordnet først,
    og deretter ordne detaljene
- Å hente data fra filer er gøy!
  - Kan jobbe med store og reelle data, uten tasting
- Når alt dere har lært frem til nå kobles sammen,
  kan det brukes til å utrette mye