

Kap. 6 ++: Brukerinput, feilhåndtering, testfunksjoner

Joakim Sundnes^{1,2}

Simula Research Laboratory¹

Universitetet i Oslo, Institutt for informatikk²

Sep 10, 2026

Plan for uke 37

Tirsdag 8/9

- Bruker-input:
 - Kommandolinjeargumenter
 - Lesing fra fil
- Oppgave 4.2, 4.3, 4.4 (+ evt 4.1)
- Feilhåndtering med try-except
- Oppgave 4.5

Torsdag 10/9

- (Mer om exceptions, ikke pensum)
- Quiz om funksjoner
- Testfunksjoner (enhetstesting)
- Oppgave 3.6, 3.7, 3.8
- (Kanskje oppg 2.19, 2.20, 2.21)
- (Lage egne modeuler, ikke pensum)

- Vi har sett hvordan vi kunne bruke funksjonen `input()` til å lese inn data fra brukeren av programmet. Denne er enkel å bruke, men blir fort upraktisk og treg.
- To nyttige alternativer:
 - Kommandolinjeargumenter
 - Lesing av data fra fil

Repetisjon: `input()`

Eksempelprogram:

```
import numpy as np

h = input('Skriv inn høyden (i meter):')
h = float(h)

p0 = 100.0      #trykk ved havnivå (kPa)
h0 = 8400       #skalahoøyde (m)

p = p0 * np.exp(-h/h0)
print(p)
```

Kjører i et terminalvindu:

```
Terminal> python altitude.py
Skriv inn høyden (i meter): 2469
74.53297273796525
```

Mer effektivt: Kommandolinjeargumenter

Eksempel på kommandolinjeargumenter:

```
Terminal> python myprog.py arg1 arg2 arg3 ...  
Terminal> cp -r dinmappe ../minmappe  
Terminal> ls -l
```

Unix-programmer (rm, ls, cp, ...) bruker i stor grad kommandolinjeargumenter, (se f.eks. man ls). Vi skal gjøre det samme.

Kommandolinjeargumenter i eksempelprogrammet

Vi ønsker å spesifisere h som et *kommandolinjeargument* etter navnet på programmet når vi kjører det:

```
Terminal> python altitude_cml.py 2469  
74.53297273796525
```

Kommandolinjeargumenter er "ordene" etter programnavnet, og disse lagres automatisk i en liste med navn `argv` i modulen `sys`.

```
import sys  
import numpy as np  
  
h = float(sys.argv[1])  
  
p0 = 100.0      #trykk ved havnivå (kPa)  
h0 = 8400       #skalahøyde (m)  
  
p = p0 * np.exp(-h/h0)  
print(p)
```

Kommandolinjeargumenter er atskilt med blanke tegn

Her er et annet program `print_cml.py`:

```
import sys; print(sys.argv)
```

Demonstrasjoner:

```
Terminal> python print_cml.py 1 2 3  
['print_cml.py', '1', '2', '3']
```

```
Terminal> python print_cml.py streng med blanke  
['print_cml.py', 'streng', 'med', 'blanke']
```

```
Terminal> python print_cml.py "streng med blanke"  
['print_cml.py', 'streng med blanke']
```

Merk 1: bruk anførselstegn, som i "streng med blanke", for å overstyre regelen at kommandolinjeargumenter er atskilt med blanke tegn.

Merk 2: alle listeelementer er omgitt av anførselstegn, noe som demonstrerer at kommandolinjeargumenter er strenger.

Lesing av data fra filer

Data som brukes i beregninger og analyse lagres vanligvis i filer. Vi trenger å kunne lese data fra filer og lagre dem i passende datastrukturer (typisk tall og lister med tall).

Eksempel på en datafil

```
21.8  
18.1  
19  
23  
26  
17.8
```

Ett tall på hver linje. Hvordan kan vi lese disse tallene?

Standardmetoden - lese en fil linje for linje

To alternativer for å åpne en fil:

```
infile = open('data.txt', 'r')    # åpne fil
for line in infile:
    # gjør noe med linjen
infile.close()                    # lukk fil
```

Mer moderne, med en with-setning:

```
with open('data.txt', 'r') as infile:
    for line in infile:
        # behandle linje
```

Merk:

- All kode for behandling av filen er en innrykket blokk
- Filen lukkes automatisk

Lese og behandle eksempelfilen over

Beregn gjennomsnittsverdiene av tallene i filen:

```
with open('data.txt', 'r') as infile:
    mean = 0
    lines = 0
    for line in infile:
        number = float(line)    #Viktig!
        mean = mean + number
        lines += 1
mean = mean/lines
print(mean)
```

Alternative måter å lese en fil på

Les alle linjer på en gang inn i en liste med strenger (linjer):

```
infile = open("filename.txt")  
lines = infile.readlines()  
infile.close()
```

```
for line in lines:  
    # behandle linje
```

Lesing av hele filen inn i en streng:

```
text = infile.read()
```

De fleste datafiler inneholder tekst blandet med tall

Fil med data om nedbør:

Gjennomsnittlig nedbør (i mm) i Roma mellom 1782 og 1970

Jan 81.2

Feb 63.2

Mar 70.3

Apr 55.7

May 53.0

Jun 36.4

Jul 17.5

Aug 27.5

Sep 60.9

Oct 117.7

Nov 111.0

Dec 97.9

Year 792.9

Hvordan leser vi en slik fil?

Behandling av hver linje med `split()`

- Hovedideen for å behandle hver linje er å dele opp linjen i ord, og prosessere hvert ord for seg (typisk konvertere tall til float)
- Pythons `split`-metode er ekstremt nyttig for å dele opp strenger

Standard oppskrift:

```
months = []
values = []
infile.readline()
for line in infile:
    words = line.split() # split i ord
    if words[0] != 'Year':
        months.append(words[0])
        values.append(float(words[1]))
```

Bli kjent med `split()`-metoden!

- Som standard vil `split()` splitte ord atskilt med mellomrom
- Vi kan spesifisere hvilken som helst streng `s` som skilletegn:
`split(s)`

```
>>> line = 'Okt 117.7'
>>> words = line.split()
>>> words
>>> ['Okt', '117.7', '']
>>> type(words[1]) # streng, ikke et tall!
>>> <type 'str'>
>>> line2 = 'output;fra;excel'
>>> line2.split(';')
>>> ['output', 'fra', 'excel']
```

Komplett program for lesing av nedbørsdata

```
with open("precip_roma.txt") as infile:
    months = []
    rainfall = []
    infile.readline() # hopp over første linje
    for line in infile:
        words = line.split()
        # words[0]: måned, words[1]: nedbør
        months.append(words[0])
        rainfall.append(float(words[1]))
    months = months[:-1] # Dropp "Year"
    annual_avg = rainfall[-1] # Lagre årsgjennomsnittet
    rainfall = rainfall[:-1] # Fjern fra listen
```

Skriving av data til fil

Veldig likt som å lese fra fil:

```
with open(filename, 'w') as outfile: # 'w' for write
    for data in somelist:
        outfile.write(sometext + '\n')
```

Hvis vi ønsker å *legge til* tekst i en fil bruker vi `open(filename, 'a')`, hvor *a* står for *append*.

Eksempel: Skriv en nøstet liste med tall til fil

Problem:

Vi har en nøstet liste (rader og kolonner):

```
data =  
[[ 0.75,      0.29619813, -0.29619813, -0.75      ],  
 [ 0.29619813,  0.11697778, -0.11697778, -0.29619813],  
 [-0.29619813, -0.11697778,  0.11697778,  0.29619813],  
 [-0.75,      -0.29619813,  0.29619813,  0.75      ]]
```

Skriv disse dataene til fil i tabellform:

Løsning:

```
with open('tmp_table.dat', 'w') as outfile:  
    for row in data:  
        for column in row:  
            outfile.write(f'{column:14.8f}')  
        outfile.write('\n')
```

Håndtering av feil i brukerinput

- Data blir ofte tastet inn feil
- Filer kan inneholde feil

Vi bør håndtere slike feil, i stedet for å la programmet stoppe med en standard Python-feilmelding

Tilbake til eksempelet fra tidligere

Kode:

```
import sys
from math import exp

h = float(sys.argv[1])

p0 = 100.0      #trykk ved havnivå (kPa)
h0 = 8400       #skal høyde (m)

p = p0 * exp(-h/h0)
print(p)
```

Hvilke feil kan oppstå her?

Feil nr 1: kommandolinjeargumentet mangler

Eksempel på bruk av programmet:

```
Terminal> python altitude.py
Traceback (most recent call last):
File "altitude.py", line 2, in ?
h = float(sys.argv[1])
IndexError: list index out of range
```

Hva skjer her?

- 1 Brukeren glemte å oppgi et kommandolinjeargument
- 2 `sys.argv` har da bare ett element, `sys.argv[0]`, som er programnavnet (`altitude.py`)
- 3 Indeks 1 finnes ikke i listen `sys.argv[1]`, og vi får derfor en `IndexError`

Feil i input kan håndteres

Hvordan kan vi ta kontroll, forklare hva som var galt med input, og stoppe programmet uten merkelige Python-feilmeldinger?

Enkel løsning med en if-test:

```
# Program altitude_if.py
import sys
if len(sys.argv) < 2:
    print('Det mangler et kommandolinjeargument.')
    exit() # avbryt
...
```

```
Terminal> python altitude.py
Det mangler et kommandolinjeargument.
```

Unntak/exceptions som et alternativ til if-tester

- If-testen i forrige eksempel testet om noe var galt, håndterte feilen hvis det var det, og gjorde ellers det vi hadde tenkt å gjøre.
- Det blir fort tungvint å skrive if-tester for alt vi tenker kan gå galt i koden
- Det er vanlig i Python (og mange andre språk) å *prøve* å gjøre det vi hadde tenkt, og hvis det feiler, gjenoppretter vi fra feilen
- Dette prinsippet bruker en try-except-blokk

```
try:  
    <setninger vi har tenkt å gjøre>  
except:  
    <setninger for håndtering av feil>
```

Hvis noe går galt i try-blokken, utløser Python et *unntak* / *exception* og programmet hopper umiddelbart til except-blokken.

Temperaturkonverteringsprogrammet med try-except

Prøv å lese h fra kommandolinjen. Hvis det feiler, fortell brukeren hva som er galt og stopp programmet.

```
import sys
try:
    h = float(sys.argv[1])
except:
    print('Det mangler et kommandolinjeargument.')
    exit()

p0 = 100.0; h0 = 8400
print(p0 * exp(-h/h0))
```

Bruk av programmet:

```
Terminal> python altitude_cml_except1.py
Det mangler et kommandolinjeargument.
```

```
Terminal> python altitude_cml_except1.py 2469m
Det mangler et kommandolinjeargument.
```

Vi kan teste for spesifikke unntak

Det er god programmeringsstil å teste for spesifikke unntak:

```
try:
    h = float(sys.argv[1])
except IndexError:
    print('Det mangler et kommandolinjeargument.')
```

Hvis vi har en indeks utenfor grensene i `sys.argv`, blir et `IndexError`-unntak utløst, og vi hopper til `except`-blokken. Hvis et annet unntak oppstår stopper programmet med en standard feilmelding:

```
Terminal> python altitude_cml_except1.py 2469m
Traceback (most recent call last):
File "altitude.py", line 3, in <module>
C = float(sys.argv[1])
ValueError: invalid literal for float(): 2469m
```

Forbedring: test både for IndexError og ValueError

```
import sys
try:
    h = float(sys.argv[1])
except IndexError:
    print('Det mangler et kommandolinjeargument for h.')
    exit()
except ValueError:
    print(f'h må være et rent tall, ikke {sys.argv[1]}')
    exit()

p0 = 100.0; h0 = 8400
print(p0 * exp(-h/h0))
```

Bruk:

```
Terminal> python altitude.py
Det mangler et kommandolinjeargument for h.
```

```
Terminal> python altitude.py 2469m
h må være et rent tall, ikke 2469m
```

Programmereren kan utløse unntak (selvstudium, ikke pensum)

- I stedet for bare å la Python utløse unntak, kan vi utløse våre egne og tilpasse meldingen til det aktuelle problemet
- Vi gir to eksempler på dette:
 - fange et unntak, men utløse et nytt med en forbedret (tilpasset) feilmelding
 - utløse et unntak på grunn av feil inputdata
- Grunnleggende syntaks: `raise ExceptionType(message)`

Eksempler på å utløse unntak på nytt med bedre meldinger

```
def read_altitude():  
    try:  
        h = float(sys.argv[1])  
    except IndexError:  
        # utløps på nytt, men med spesifikk forklaring:  
        raise IndexError('Høyden må oppgis på kommandolinjen.')  
    except ValueError:  
        # utløps på nytt, men med spesifikk forklaring:  
        raise ValueError(f'h må være et tall, ikke {sys.argv[1]}.'.)  
  
    # h er lest riktig som et tall, men har en feil verdi:  
    if h < -430 or h > 13000:  
        raise ValueError(f'Formelen er ikke gyldig for h={h}')
```

return h

Kaller den forrige funksjonen og kjører programmet

```
try:
    h = read_altitude()
except (IndexError, ValueError) as e:
    # skriv ut unntaksmeldingen og stopp programmet
    print(e)
    exit()
```

Implementasjon:

Terminal> python altitude_cml_except2.py
Høyden må oppgis på kommandolinjen.

Terminal> python altitude_cml_except2.py 1000m
Høyden må være et tall, ikke 1000m.

Terminal> python altitude_cml_except2.py 20000
Formelen er ikke gyldig for h=20000.

Terminal> python altitude_cml_except2.py 8848
34.8773231887747

Hvordan kan vi sikre at et program fungerer som det skal?

- Test alle mulige bruksområder av programmet, og verifiser at resultatet er som forventet (vanskelig/umulig)
- Test mindre deler (f.eks. funksjoner) for noen få velvalgte bruksområder (enkelt)
- Den andre prosedyren for å verifisere programkode kalles *enhetstesting*

Enhetstesting er "første linje" i programvaretesting

Testing i større programsystemer gjøres kontinuerlig, enten med faste tidsintervaller eller ved at testene automatisk kjøres ved spesielle operasjoner.

- Enhetstesting: Mange små enkelttester som kjøres ofte og helautomatisk. Kjøres automatisk, gjerne daglig eller hver gang koden endres.
- Regresjonstesting: Tester større deler av programmet, og sammenlikner output med output fra en tidligere versjon.
- Arbeidsflyt-testing (work flow tests): Tester større operasjoner og hvordan brukere typisk bruker programmet.

- En funksjon vil ofte bli kalt mange ganger
- For å sikre at funksjonen virker som vi har tenkt, kan vi skrive spesielle *testfunksjoner*
- Helst bør alle funksjoner ha én eller flere tilhørende testfunksjon(er)
- En testfunksjon skrives ofte før selve funksjonen! (testdrevet utvikling)

Eksempel på en testfunksjon

```
def square(x):  
    return x**2  
  
def test_square():  
    x = 4                                # en valgt x-verdi  
    expected = 16                        # forventet resultat fra double(x)  
    computed = square(x)  
    success = computed == expected      # boolsk verdi: test bestått?  
    msg = f'beregnet {computed}, forventet {expected}'  
    assert success, msg
```

Regler for testfunksjoner:

- Navnet begynner alltid med test_
- Testfunksjoner tar ingen argumenter
- Testfunksjoner må ha en assert success-setning, der success er True hvis testen er bestått og False ellers

Bruk av en spesifikk melding (msg) er valgfritt. Både assert success og assert success, msg fungerer, men den siste vil skrive ut msg ved feil

Hvorfor skrive testfunksjoner som følger disse reglene?

- Enkelt å gjenkjenne hva som er testfunksjoner og hva som er "vanlige" funksjoner
- Testrammeverk som pytest kan automatisk kjøre *alle* testfunksjonene dine (i en mappestruktur) og rapportere om det har sneket seg inn noen feil
- Det er en veletablert standard

En testfunksjon kan inneholde flere tester

```
def square(x):                # en eller annen funksjon
    return x ** 2

def test_square():            # tilhørende testfunksjon
    tol = 1E-14                # toleranse for sammenligning av flyttall
    x_values = [3, 7, -2, 0, 4.5]
    expected_values = [9, 49, 4, 0, 20.25]
    for x, expected in zip(x_values, expected_values):
        computed = square(x)
        msg = f'{computed} != {expected}'
        assert abs(expected - computed) < tol, msg
```

- Hvis en av testene feiler vil assert utløse en AssertionError
- Hvis alle testene passerer skjer det ingenting; testfunksjonen kjører *stille*.

Kommentarer om testfunksjoner

- Mange synes testfunksjoner er et vanskelig tema
- Ideen er enkel: Lag et problem der du vet svaret, kall funksjonen, sammenlign resultatet med det kjente svaret
- Øvelse hjelper for å lære konseptet - som alltid

Det kan være forvirrende at en testfunksjon som passerer kjører stille. Under utvikling (og mens du lærer) kan det være praktisk å sette inn noen print-setninger slik at du ser at testfunksjonene blir kjørt. Alternativt, bruk et verktøy som pytest.

Lage egne moduler (selvstudium, ikke pensum)

Vi har ofte brukt moduler som `math` og `sys`:

```
from math import log
r = log(6)      # kall log-funksjonen i math-modulen

import sys
x = eval(sys.argv[1]) # få tilgang til listen argv i sys-modulen
```

Karakteristika for moduler:

- Samling av nyttige data og funksjoner
- Funksjoner i en modul kan gjenbrukes i mange forskjellige programmer
- Hvis du har noen generelle funksjoner som kan være nyttige i mer enn ett program, lag en modul med disse funksjonene
- Det er enkelt: bare samle funksjonene du vil ha i en fil, og det er en modul!

Eksempel på å lage en egen modul

Ulike formler for renteregning:

$$A = A_0 \left(1 + \frac{p}{360 \cdot 100}\right)^n, \quad (1)$$

$$A_0 = A \left(1 + \frac{p}{360 \cdot 100}\right)^{-n}, \quad (2)$$

$$n = \frac{\ln \frac{A}{A_0}}{\ln \left(1 + \frac{p}{360 \cdot 100}\right)}, \quad (3)$$

$$p = 360 \cdot 100 \left(\left(\frac{A}{A_0} \right)^{1/n} - 1 \right). \quad (4)$$

A_0 : startbeløp, p : prosentandel, n : dager, A : sluttbeløp

Vi vil lage en modul med disse fire funksjonene.

Først lager vi Python-funksjoner for formlene

```
from math import log as ln

def present_amount(A0, p, n):
    return A0*(1 + p/(360.0*100))**n

def initial_amount(A, p, n):
    return A*(1 + p/(360.0*100))**(-n)

def days(A0, A, p):
    return ln(A/A0)/ln(1 + p/(360.0*100))

def annual_rate(A0, A, n):
    return 360*100*((A/A0)**(1.0/n) - 1)
```

Så kan vi lage modulfilen

- Samle de 4 funksjonene i en fil `interest.py`
- Nå er `interest.py` faktisk en modul `interest` (!)

Eksempel på bruk. Hvor lang tid tar det å doble et beløp?

```
from interest import days
A0 = 1; A = 2; p = 5
n = days(A0, A, p)
years = n / 365.0
print(f'Pengene har doblet seg etter {years:.1f} år')
```

Legge til en testblokk i en modulfil

- Modulfiler kan ha en if-test på slutten som inneholder en *testblokk* for å teste eller demonstrere modulen
- Testblokken blir ikke utført når filen importeres som en modul i et annet program
- Testblokken blir utført *bare* når filen kjøres som et selvstendig program

```
if __name__ == '__main__': # denne testen definerer testblokken  
    <blokk med setninger>
```

I vårt tilfelle:

```
if __name__ == '__main__':  
    A = 2.2133983053266699  
    A0 = 2.0  
    p = 5  
    n = 730  
    A_ = present_amount(A0, p, n)  
    A0_ = initial_amount(A, p, n)  
    d_ = days(A0, A, p)  
    p_ = annual_rate(A0, A, n)  
    print(f'A={A_} ({A}) A0={A0_} ({A}) n={n_} ({n}) p={p_} ({p})')
```

Hvordan kan Python finne vår nye modul?

- Alt 1: Modulfil og hovedprogram er i samme mappe
- Alt 2: Legg egne hjemmelagde moduler i en egen mappe
 - Må legges til i listen over mapper hvor Python leter
- Alt 3: Lag en egen pakke som kan installeres med `pip install`

I IN1900 bruker vi bare alternativ 1. Alternativ 3 anbefales for de som vil lage en "proff" løsning, både for eget bruk og som eventuelt kan deles med andre.