

Kap. 3: Løkker og lister

Joakim Sundnes^{1,2}

Simula Research Laboratory¹

Universitetet i Oslo, Institutt for informatikk²

Aug 27, 2026

- Kort quiz om temaene fra forrige uke
- Oppgave 1.12 fra *Primer on Scientific Programming with Python*
- Løkker og lister (Kap 3.2-3.6):
 - While-løkker
 - Boolske uttrykk (True/False)
 - For-løkker
- Oppgave 2.1, 2.3, 2.4, 2.7, 2.8 og 2.9 fra *Primer on Scientific Programming with Python*
- Oppgave 5.1 fra *Oppgaveheftet*
- Vilkår (if-tester, kap 3.1)
- Mer om løkker og lister (sannsynligvis neste uke, delvis selvstudium):
 - range, zip, listekomprehensjon (list comprehensions),...

Spørsmål 1

Hvilke av følgende kodesegmenter vil gi en feilmelding når de kjøres (hvis noen)? Hva er feil?

Kode 1

```
a = "Hello world"  
a = 2
```

Kode 2

```
pi = 3.14159  
pi = 2 * pi
```

Kode 3

```
b = x**2 + 3 * x + 1  
x = 2.3
```

Svar på spørsmål 1

```
#kode 1
```

```
a = "Hello world"
```

```
a = 2
```

```
#kode 2
```

```
pi = 3.14159
```

```
pi = 2 * pi
```

```
#kode 3
```

```
b = x**2 + 3 * x + 1
```

```
x = 2.3
```

```
Terminal> python quiz1.py
```

```
Traceback (most recent call last):
```

```
File "quiz1.py", line 10, in
```

```
b = x**2 + 3 * x + 1
```

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

I programmering må variabler defineres før de brukes (i motsetning til i matematikk).

Hvilke typer har variablene i følgende kode:

```
a = 2  
b = 2.5  
s = "hello"  
t = a * s
```

Svar på spørsmål 2

```
a = 2
b = 2.5
s = "hello"
t = a * s

print('a is ', type(a), ' b is ', type(b))
print(' s is ', type(s), ' t is ', type(t))
```

```
Terminal> python quiz2.py
a is <class 'int'>  b is <class 'float'>
s is <class 'str'>  t is <class 'str'>
```

Hvilke av disse kodene er feil (hvis noen)?

```
from numpy import sin, pi  
x = sin(pi / 2)
```

```
import numpy  
x = sin(pi / 2)
```

```
from numpy import *  
x = sin(pi / 2)
```

Svar på spørsmål 3

Den andre import-koden er feil. Hvis du importerer en modul på denne måten, må alle funksjoner og variabler fra modulen ha modulnavnet som prefiks:

```
import numpy
x = numpy.sin(numpy.pi / 2)
```

Den tredje import-koden er riktig, men er regnet som dårlig kodelstil. Standard-måten å importere numpy er

```
import numpy as np
x = np.sin(np.pi / 2)
```

Spørsmål 4 (diskusjon)

I Python kan vi lage og senere endre en variabel slik:

```
a = 2  
(...)  
a = 0.5*2
```

I mange andre programmeringsspråk må vi skrive noe sånt som:

```
int a;  
a = 2;  
(...)
```

Python sparer åpenbart litt skrijving, men kan du tenke deg noen potensielle problemer med Pythons måte å definere variabler på?

Svar på spørsmål 5

Dette enkle eksempelet illustrerer en potensiell fallgrube med dynamisk typing:

```
#lag en variabel x0:  
x0 = 10.0
```

```
#Gjør noe nyttig...
```

```
#endre x0:  
x0 = 14.0
```

```
#Flere viktige beregninger med x0...  
print('The value of x0 is ', x0)
```

Bruk av løkker for å gjenta lignende operasjoner:

- while-løkker (vilkårsløkker)
 - Boolske uttrykk (True/False)
- for-løkker (telleløkker)
 - Lister

Eksempel: tabell med Celsius- og Fahrenheit-grader

-20	-4.0
-15	5.0
-10	14.0
-5	23.0
0	32.0
5	41.0
10	50.0
15	59.0
20	68.0
25	77.0
30	86.0
35	95.0
40	104.0

Hvordan kan et program skrive ut en slik tabell?

Lage en tabell: den enkle, naive løsningen

Vi vet hvordan vi lager én linje i tabellen:

```
C = -20  
F = 9.0 / 5 * C + 32  
print(C, F)
```

Vi kan bare gjenta disse instruksjonene:

```
C = -20; F = 9.0 / 5 * C + 32; print(C, F)  
C = -15; F = 9.0 / 5 * C + 32; print(C, F)  
...  
  
C = 35; F = 9.0 / 5 * C + 32; print(C, F)  
C = 40; F = 9.0 / 5 * C + 32; print(C, F)
```

- Veldig kjedelig å skrive, lett å introdusere en skrivefeil
- Når programmering blir kjedelig, finnes det vanligvis en konstruksjon som automatiserer oppgaven!
- Datamaskinen er ekstremt god til å utføre repetitive oppgaver
- Til dette formålet bruker vi *løkker* (loops)

while-løkken gjør det mulig å gjenta lignende oppgaver

En while-løkke utfører gjentatte ganger et sett med instruksjoner så lenge et vilkår er oppfylt. (kalles også en vilkårsløkke)

```
while betingelse:  
    <instruksjon 1>  
    <instruksjon 2>  
    ...  
<første instruksjon etter løkken>
```

- Alle instruksjoner i løkken må rykkes inn (indenteres)!
- Løkken avsluttes når en ikke-innrykket instruksjon oppdages

Eksempel 1: tabell med while-løkke

while-løkken er en mye mer effektiv måte å lage Fahrenheit-Celcius-tabellen beskrevet på de forrige lysbildene.

Oppgave: Gitt et intervall av Celsius-grader fra -20 til 40, i trinn på 5, beregn de tilsvarende Fahrenheit-gradene og skriv ut begge verdiene til skjermen.

while-løkken for å lage en tabell

```
print('-----') # tabelloverskrift
C = -20           # startverdi for C
dC = 5            # økning av C i løkken
while C <= 40:    # løkkehode med betingelse
    F = (9.0 / 5) * C + 32 # 1. instruksjon inne i løkken
    print(C, F)           # 2. instruksjon inne i løkken
    C = C + dC            # siste instruksjon inne i løkken
print('-----') # slutt på tabellinjen
```

Programflyten i en while-løkke

La oss simulere while-løkken for hånd:

- Først er $C = -20$, $-20 \leq 40$ er sant, derfor utfører vi løkkeinstruksjonene
- Beregn F , skriv ut, og oppdater C til -15
- Vi hopper opp til `while`-linjen, evaluerer $C \leq 40$, som er sant, derav en ny runde i løkken
- Vi fortsetter på denne måten til C er oppdatert til 45
- Nå er løkkebetingelsen $45 \leq 40$ usann, og programmet hopper til den første linjen etter løkken - løkken er over

Boolske uttrykk er sanne eller usanne

Et uttrykk med verdien sann eller usann kalles et boolsk uttrykk.

Eksempler: $C = 40$, $C \neq 40$, $C \geq 40$, $C > 40$, $C < 40$.

```
C == 40    # merk dobbel ==, C = 40 er en tilordning!  
C != 40  
C >= 40  
C > 40  
C < 40
```

Vi kan teste boolske uttrykk i en interaktiv Python-sesjon:

```
>>> C = 41  
>>> C != 40  
>>> True  
>>> C < 40  
>>> False  
>>> C == 41  
>>> True
```

Kombinere boolske uttrykk

Flere betingelser kan kombineres med and/or (og/eller):

```
while betingelse1 and betingelse2:
...

while betingelse1 or betingelse2:
...
```

Regel 1: C1 and C2 er True hvis både C1 og C2 er True

Regel 2: C1 or C2 er True hvis én av C1 eller C2 er True

```
>>> x = 0; y = 1.2
>>> x >= 0 and y < 1
>>> False
>>> x >= 0 or y < 1
>>> True
>>> x > 0 or y > 1
>>> True
>>> x > 0 or not y > 1
>>> False
>>> -1 < x <= 0    # -1 < x and x <= 0
>>> True
>>> not (x > 0 or y > 0)
>>> False
```

Lister er objekter for å lagre en sekvens av ting (objekter)

Så langt har én variabel referert til ett tall (eller en streng), men noen ganger har vi naturligvis en samling av tall, for eksempel gradene $-20, -15, -10, -5, 0, \dots, 40$

Enkel løsning: én variabel for hver verdi

C1 = -20

C2 = -15

C3 = -10

...

C13 = 40

Dum og kjedelig løsning hvis vi har mange verdier!

Bedre: et sett med verdier kan samles i en liste

C = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]

Nå er det én variabel, C, som inneholder alle verdiene

Listeoperasjoner 1: initialisering og indeksering

Initialiser med firkantparenteser og komma mellom Python-objektene:

```
L1 = [-91, 'en streng', 7.2, 0]
```

Elementer aksesseres via en indeks: `L1[3]` (indeks=3).

Listeindekser starter på 0: 0, 1, 2, ... `len(L1)-1`.

```
>>> mylist = [4, 6, -3.5]
>>> print(mylist[0])
>>> 4
>>> print(mylist[1])
>>> 6
>>> print(mylist[2])
>>> -3.5
>>> len(mylist)    # lengden av listen
>>> 3
```

Listeoperasjoner 2: append, pluss, lengde

```
>>> C = [-10, -5, 0, 5, 10, 15, 20, 25, 30]
>>> C.append(35)    # legg til nytt element 35 på slutten
>>> C
>>> [-10, -5, 0, 5, 10, 15, 20, 25, 30, 35]
>>> C = C + [40, 45]    # utvid C på slutten
>>> len(C)             # lengden av listen
>>> 12
```

for-løkken brukes for å iterere over en liste

En for-løkke itererer over elementene i en liste, og utfører operasjoner på hver av dem:

```
for element in list:  
    <instruksjon 1>  
    <instruksjon 2>  
    ...  
<første instruksjon etter løkken>
```

- Enklere enn while-løkken (ingen betingelse nødvendig)
- Litt mindre fleksibel

Eksempel: for-løkke for temperaturkonvertering

Oppgave: Lag en liste med Celsius-verdier tilsvarende den forrige. Bruk en for-løkke for å iterere over listen, beregn de tilsvarende Fahrenheit-verdiene og skriv ut verdiene til skjermen.

Gå gjennom elementer i en liste med en for-løkke

Bruk en *for*-løkke til å gå gjennom en liste og behandle hvert element:

```
degrees = [0, 10, 20, 40, 100]
for C in degrees:
    print('Celsius-grader:', C)
    F = 9/5.*C + 32
    print('Fahrenheit:', F)
print('Grader-listen har', len(degrees), 'elementer')
```

Akkurat som med *while*-løkker må instruksjonene i løkken rykkes inn (indenteres)!

Lage en tabell med en for-løkke

Tabell med Celsius- og Fahrenheit-grader:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
for C in Cdegrees:
    F = (9.0/5)*C + 32
    print(C, F)
```

Merk: `print(C, F)` gir en stygg utskrift. Bruk en f-streng for å formatere de to kolonnene pent:

```
print(f'{C:5d} {F:5.1f}')
```

Utskrift:

```
-20  -4.0
-15   5.0
-10  14.0
-5   23.0
0    32.0
.....
35   95.0
40  104.0
```

En for-løkke kan alltid oversettes til en while-løkke

for-løkken

```
for element in somelist:  
    # behandle elementet
```

kan alltid transformeres til en tilsvarende while-løkke

```
index = 0  
while index < len(somelist):  
    element = somelist[index]  
    # behandle elementet  
    index += 1
```

Men ikke alle while-løkker kan uttrykkes som for-løkker!

Lagre tabellkolonnene som lister

La oss legge alle Fahrenheit-verdiene i en liste også:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
Fdegrees = [] # start med tom liste
for C in Cdegrees:
    F = (9.0/5)*C + 32
    Fdegrees.append(F) # legg til nytt element i Fdegrees
print(Fdegrees)
```

print(Fdegrees) resulterer i

```
[-4.0, 5.0, 14.0, 23.0, 32.0, 41.0, 50.0, 59.0,
68.0, 77.0, 86.0, 95.0, 104.0]
```

Bruke range for å iterere over indekser

Noen ganger har vi ikke en liste, men ønsker å gjenta en operasjon N ganger. Python-funksjonen `range` returnerer en liste med heltall:

```
C = 0
for i in range(N):
    F = (9.0/5) * C + 32
    print(F)
```

- `range(start, stop, inc)` genererer en liste med heltall `start`, `start+inc`, `start+2*inc`, og så videre opp til, *men ikke inkludert*, `stop`.
- `range(stop)` er en forkortelse for `range(0, stop, 1)`.

(I Python 3 returnerer `range` en *iterator*, som strengt tatt ikke er en liste, men oppfører seg på samme måte når den brukes i en `for`-løkke.)

Implementere en matematisk sum via en løkke

$$S = \sum_{i=1}^N i^2$$

```
N = 14  
  
S = 0  
for i in range(1, N+1):  
    S += i**2
```

Eller (mindre vanlig):

```
S = 0  
i = 1  
while i <= N:  
    S += i**2  
    i += 1
```

Matematiske summer dukker opp ofte, så det er verdt å huske denne oppskriften!

- Endring av lister i en for-løkke
- Oppgave 2.4, 2.7, 2.8 og 2.9 fra *Primer on Scientific Programming with Python*
- Oppgave 5.1 fra *Oppgaveheftet*
- Vilkår (if-tester, kap 3.1)
- Mer om if-tester, løkker og lister (noe neste uke/selvstudium)
 - Kortversjoner av løkker og if-tester (list comprehension, inline if-tester)
 - Traversering av flere lister med zip
 - Nøstede lister, nøstede løkker
 - Tupler (konstante lister)

Hvordan kan vi endre elementene i en liste?

La oss si at vi vil legge til 2 til alle tallene i en liste:

```
v = [-1, 1, 10]
for e in v:
    e = e + 2
print(v)
```

```
[-1, 1, 10]    # uendret!!
```

Endring av et listelement krever tilordning til et indeksert element

Hva er problemet?

Inne i løkken er `e` en vanlig (`int`) variabel, som er en *kopi* av elementene i listen. Første gang blir `e` 1, neste gang blir `e` 3, og deretter 12, men listen `v` er uendret.

Løsning: må *indeksere et listelement* for å endre verdien:

```
v = [-1, 1, 10]
for i in range(len(v)):
    v[i] = v[i] + 2
print(v)
```

Boolske variabler og if-tester (1)

Vi husker fra tidligere at while-løkker gjentar et sett med operasjoner så lenge en betingelse er oppfylt:

```
odd = 1
while odd < 100:
    print(odd)
    odd += 2
```

Betingelsen her er et uttrykk som returnerer en boolsk variabel.

Boolske variabler og if-tester (2)

En annen viktig bruk av vilkår og boolske variabler er når vi ønsker å utføre ulike handlinger avhengig av en betingelse:

Eksempel, stykkvis konstant funksjon:

$$f(x) = \begin{cases} \sin(x), & 0 \leq x \leq \pi \\ 0, & \text{ellers} \end{cases}$$

En Python-implementering av f må teste verdien til x og gjøre to utregninger:

```
from numpy import sin, pi

x = input('Skriv inn verdien til x:')
x = float(x)

if 0 <= x <= pi:
    f = sin(x)
else:
    f = 0

print(f)
```

Den generelle formen for if-tester

if-else (else-blokken kan droppes):

```
if betingelse:  
    <kodeblokk som utføres hvis betingelsen er oppfylt>  
else:  
    <kodeblokk som utføres hvis betingelsen ikke er oppfylt>
```

Flere if-else i samme blokk:

```
if betingelse1:  
    <kodeblokk>  
elif betingelse2:  
    <kodeblokk>  
elif betingelse3:  
    <kodeblokk>  
else:  
    <kodeblokk>  
<neste setning>
```

Eksempel på flere forgreninger

En stykkvis definert funksjon

$$N(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \leq x < 1 \\ 2 - x, & 1 \leq x < 2 \\ 0, & x \geq 2 \end{cases}$$

Python-implementering med flere if-else-forgreninger

```
x = float(input("Skriv inn verdien til x:"))
if x < 0:
    print(0)
elif 0 <= x < 1:
    print(x)
elif 1 <= x < 2:
    print(2 - x)
elif x >= 2:
    print(0)
```

Veldig vanlige oppgaver i et python-program:

- Lag en liste og fyll den med verdier
- Sett en verdi til en variabel basert på en betingelse

Begge oppgaver gjøres så ofte at det er laget mer kompakte måter å gjøre det på; *list comprehensions* og *inline if-tester*.

Vanlig konstruksjon

```
odd = []  
for i in range(1,100,2):  
    odd.append(i)
```

Kortversjon med list comprehension:

```
odd = [i for i in range(1,100,2)]
```

Generell form for list comprehension:

```
newlist = [expression for element in somelist]
```

hvor expression er et uttrykk som (vanligvis) inkluderer element

Inline if-tester for kortere kode

Vanlig konstruksjon:

```
if betingelse:
    variabel = verdi1
else:
    variabel = verdi2
```

Mer kompakt syntaks med enlinjes if-else:

```
variabel = (verdi1 if betingelse else verdi2)
```

Eksempel:

```
from numpy import sin
x = float(input("Skriv inn x:"))
f = (sin(x) if 0 <= x <= pi else 0)
print(f)
```

Gjennomgå flere lister samtidig med zip

Kan vi ha én løkke som går gjennom to lister?

Løsning 1: iterere over indekser

```
for i in range(len(Cdegrees)):  
    print(Cdegrees[i], Fdegrees[i])
```

Løsning 2: bruke zip-konstruksjonen (mer "pythonsk"):

```
for C, F in zip(Cdegrees, Fdegrees):  
    print(C, F)
```

Eksempel med tre lister:

```
l1 = [3, 6, 1]; l2 = [1.5, 1, 0]; l3 = [9.1, 3, 2]  
for e1, e2, e3 in zip(l1, l2, l3):  
    print(e1, e2, e3)
```

Spørsmål: Hva skrives ut?

Svar på spørsmål

```
l1 = [3, 6, 1]; l2 = [1.5, 1, 0]; l3 = [9.1, 3, 2]
for e1, e2, e3 in zip(l1, l2, l3):
    print(e1, e2, e3)
```

```
3 1.5 9.1
6 1 3
1 0 2
```

Nøstede lister: liste av lister

- En liste kan inneholde *hvilket som helst* objekt, også en annen liste
- I stedet for å lagre en tabell som to separate lister (én for hver kolonne), kan vi slå sammen de to listene i en ny liste:

```
Cdegrees = list(range(-20, 41, 5)) #range lager en iterator, konverter  
Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]
```

```
table1 = [Cdegrees, Fdegrees] # liste av to lister
```

```
print(table1[0])      # Cdegrees-listen  
print(table1[1])      # Fdegrees-listen  
print(table1[1][2])   # det 3. elementet i Fdegrees
```

Trekke ut dellister (eller utsnitt/slices)

Vi kan enkelt hente ut deler av en liste:

```
>>> A = [2, 3.5, 8, 10]
>>> A[2:]    # fra indeks 2 til slutten av listen
>>> [8, 10]
>>> A[1:3]   # fra indeks 1 opp til, men ikke inkl., indeks 3
>>> [3.5, 8]
>>> A[:3]    # fra starten opp til, men ikke inkl., indeks 3
>>> [2, 3.5, 8]
>>> A[1:-1]  # fra indeks 1 til det nest siste elementet
>>> [3.5, 8]
>>> A[:]     # hele listen
>>> [2, 3.5, 8, 10]
```

Merk: utsnitt/slices av lister er *kopier* av den opprinnelige listen!

Iterasjon over generelle nøstede lister

Liste med mange indekser: `somelist[i1][i2][i3]...`

Løkker over listeindekser:

```
for i1 in range(len(somelist)):
    for i2 in range(len(somelist[i1])):
        for i3 in range(len(somelist[i1][i2])):
            for i4 in range(len(somelist[i1][i2][i3])):
                value = somelist[i1][i2][i3][i4]
                # do something with 'value'...
```

Løkker over dellister:

```
for sublist1 in somelist:
    for sublist2 in sublist1:
        for sublist3 in sublist2:
            for sublist4 in sublist3:
                value = sublist4
                # do something with 'value'...
```

Iterasjon over en spesifikk nøstet liste

```
L = [[9, 7], [-1, 5, 6]]  
for row in L:  
    for column in row:  
        print(column)
```

Question

Spørsmål 1: Hva skrives ut? Spørsmål 2: Hvordna kan vi indeksere elementet med verdien 5? Simuler dette programmet for hånd!

Utskrift:

9
7
-1
5
6

Indeksering av verien 5:

L[1][1]

Tupler (tuples) er konstante lister

Tupler er konstante lister som ikke kan endres:

```
>>> t = (2, 4, 6, 'temp.pdf')      # definer en tuppel
>>> t = 2, 4, 6, 'temp.pdf'      # kan droppe parenteser
>>> t[1] = -1
>>> ...
>>> TypeError: object does not support item assignment
>>> t.append(0)
>>> ...
>>> AttributeError: 'tuple' object has no attribute 'append'
>>> del t[1]
>>> ...
>>> TypeError: object doesn't support item deletion
```

Tupler kan gjøre mye av det lister kan gjøre:

```
>>> t = t + (-1.0, -2.0)          # legg sammen to tupler
>>> t
>>> (2, 4, 6, 'temp.pdf', -1.0, -2.0)
>>> t[1]                          # indeksering
>>> 4
>>> t[2:]                        # deltuppel/utsnitt
>>> (6, 'temp.pdf', -1.0, -2.0)
>>> 6 in t                       # medlemskap
>>> True
```

Hvorfor tupler når lister har mer funksjonalitet?

- Tupler er konstante og dermed beskyttet mot tilfeldige endringer
- Tupler er raskere enn lister
- Tupler er mye brukt i Python-programvare

(så du må vite om dem!)

- Tupler (men ikke lister) kan brukes som nøkler i ordbøker/dictionaries

(mer om ordbøker/dictionaries senere)

Nøkkeltemaer fra dette kapitlet

- while-løkker
- Boolske uttrykk
- for-løkker
- Lister
- Nøstede lister
- Tupler

Oppsummering av løkker, lister og tupler

while-løkker og for-løkker:

```
while betingelse:
```

```
for element in somelist:
```

Lister og tupler:

```
mylist = ['en streng', 2.5, 6, 'en annen streng']  
mytuple = ('en streng', 2.5, 6, 'en annen streng')  
mylist[1] = -10  
mylist.append('en tredje streng')  
mytuple[1] = -10 # ulovlig: kan ikke endre en tuppel
```

Listeoperasjoner, selvstudium: sett inn (insert), slett (delete)

```
>>> C
>>> [-10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> C.insert(0, -15)      # sett inn -15 på indeks 0
>>> C
>>> [-15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> del C[2]              # slett det 3. elementet
>>> C
>>> [-15, -10, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> del C[2]              # slett det som nå er det 3. elementet
>>> C
>>> [-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

Listeoperasjoner, selvstudium: søk, negative indekser

```
>>> C.index(10)      # indeksen til det første elementet med verdien 10
>>> 3
>>> 10 in C          # er 10 et element i C?
>>> True
>>> C[-1]            # det siste listeelementet
>>> 45
>>> C[-2]            # det nest siste listeelementet
>>> 40
>>> somelist = ['book.tex', 'book.log', 'book.pdf']
>>> texfile, logfile, pdf = somelist # tilordne direkte til variabler
>>> texfile
>>> 'book.tex'
>>> logfile
>>> 'book.log'
>>> pdf
>>> 'book.pdf'
```

Listeoperasjoner, selvstudium: oppsummering

Konstruksjon	Betydning
<code>a = []</code>	initialiser en tom liste
<code>a = [1, 4.4, 'run.py']</code>	initialiser en liste
<code>a.append(elem)</code>	legg til objektet <code>elem</code> på slutten
<code>a + [1,3]</code>	legg sammen to lister
<code>a.insert(i, e)</code>	sett inn elementet <code>e</code> før indeks <code>i</code>
<code>a[3]</code>	indekser et listeelement
<code>a[-1]</code>	hent det siste listeelementet
<code>a[1:3]</code>	utsnitt: kopier data til delliste (indeks 1, 2)
<code>del a[3]</code>	slett et element (indeks 3)
<code>a.remove(e)</code>	fjern et element med verdien <code>e</code>
<code>a.index('run.py')</code>	finn indeksen som tilsvarer en elementsverdi
<code>'run.py' in a</code>	test om en verdi finnes i listen
<code>a.count(v)</code>	tell antall elementer som har verdien <code>v</code>
<code>len(a)</code>	antall elementer i liste <code>a</code>
<code>min(a)</code>	det minste elementet i <code>a</code>
<code>max(a)</code>	det største elementet i <code>a</code>
<code>sum(a)</code>	legg sammen alle elementene i <code>a</code>
<code>sorted(a)</code>	returner en sortert versjon av liste <code>a</code>
<code>reversed(a)</code>	returner en omvendt sortert versjon av liste <code>a</code>
<code>b[3][0][2]</code>	nøstet listeindeksering
<code>isinstance(a, list)</code>	er True hvis <code>a</code> er en liste
<code>type(a) is list</code>	er True hvis <code>a</code> er en liste