

Datastrukturer: arrays og dictionaries

Joakim Sundnes^{1,2}

Simula Research Laboratory¹

Institutt for Informatikk, Universitetet i Oslo²

Sep 15, 2026

To nye datastrukturer: Numpy-arrayer og dictionaries

- Numpy-arrayer
- Oppgave 5.1, 5.2, 5.3, 5.7, fra "A primer on..."
- Dictionaries
- Oppgave 6.7 fra "A primer on..."

NB: Ingen forelesning torsdag 17/9

Fire måter å lagre en sekvens av verdier på

- *Lister*: Kan endre innhold, operasjoner gjøres ett element av gangen.
- *Tupler*: Kan ikke endre innhold, og operasjoner gjøres ett element av gangen.
- *Arrayer*:
 - Kan endre innhold
 - Tillater kun én datatype av gangen
 - Støtter *vektoriserte* operasjoner.
- *Dictionaries*:
 - Kan endre innhold
 - Operasjoner gjøres ett element av gangen
 - Bruker *nøkler* i stedet for indekser.

Arrays og dictionaries - hva brukes de til?

Arrayer:

- Brukes til numeriske beregninger
- Mindre fleksible enn lister, men mye mer effektive
- Tillater *vektoriserte* operasjoner

Dictionaries (ordbøker):

- Fleksible
- Bruker tekst eller andre objekter som *nøkler* i stedet for indekser

Del 1: Numpy-arrayer

Vi har brukt Numpy til å regne med tall, men hovedpoenget med Numpy er å gjøre beregninger på mange tall samtidig, lagret i en array.

Ulike måter å lage en array

```
import numpy as np

# Fra en liste
x = np.array([1, 3, 7])

# Med arange (tilsvarende range)
x = np.arange(21)
y = np.arange(1, 22, 2)

# arange støtter også desimaltall som steglengde:
z = np.arange(0, 20, 0.5)
```

Viktigste Numpy-funksjoner: linspace og zeros

```
# Et grid med 41 verdier fra 0.0 til 4.0  
x = np.linspace(0, 4, 41)  
  
# Array med tusen 0'er eller 1'ere  
x = np.zeros(1000)  
x = np.ones(1000)  
  
#array med 0'er, med samme form som x:  
y = np.zeros_like(x)
```

Spesielt `np.linspace` kommer vi til å bruke ofte!

Eksempel - lag en array med tusen 3.14-verdier:

```
x = 3.14 * np.ones(1000)
```

Arrayer kan ha flere dimensjoner

Numpy-arrayer kan ha opp til 32 dimensjoner, men en-, to- og tredimensjonale arrayer er mest vanlig:

```
#2D-array (3x3-matrise)
a = np.array([[1,2,3],[4,5,6],[7,8,9]])

#3D-array (3x3x3 tensor)
b = np.zeros((3,3,3)) #argumentet er en tuppel
print(b.ndim) #output: 3
print(len(b)) #3
print(b.size) #27
print(b.shape)  #(3, 3, 3)
```

Eksempler:

- Et sort-hvitt-bilde (gråtoner) kan lagres som en 2D-array
- Et fargebilde (RGB) kan lagres som en 3D-array ($n \times m \times 3$)

Arrayer er mindre fleksible enn lister (1)

- Vi kan ha ulike typer objekter, f.eks. tall og lister i samme liste, eller lister av forskjellig lengde
- En array tillater ikke dette. Alle "sub-arrays" i en dimensjon må ha samme lengde. F.eks. har en 2D-array et fast antall rader og kolonner

#lovlig (2D-array med form 3x3):

```
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
```

#ikke lovlig (gir en feilmelding)

```
a = np.array([[1,2,3],[4,5,6],[7,8]])
```

Arrayer er mindre fleksible enn lister (2)

- Arrayer har fast form og størrelse (ingen a.append, som vi har for lister)
- Dette har en teknisk grunn: en array er lagret på ett sted i minnet, og det settes av plass til hele arrayen når den opprettes.
- Typisk bruk:
 - Lister: lag en tom liste, fyll med verdier i en for-løkke
 - Arrayer: lag en array av 0'er (f.eks. `np.zeros`), fyll verdier
- Vi *kan* endre form på en array, men det er ineffektivt og ikke anbefalt. Hvis vi trenger denne fleksibiliteten er det vanligvis bedre å bruke lister

Indeksering av arrayer

Array kan indekseres som lister:

```
#element med index 0, 1, 2:  
print(b[0][1][2])
```

```
#alternativt (anbefalt)  
print(b[0,1,2])
```

Slicing fungerer på samme måte som lister: `a[start:stop:step]` henter ut alle elementer fra og med start, opp til (men ikke inkludert) stop, med steglengde step.

```
>>> a = np.linspace(0,10,101)  
>>> a[:-1:] #alt bortsett fra siste element  
>>> a[:-1]  #samme som over  
>>> a[2::2] #hvert andre element, fra indeks 2 og opp  
>>> a[::-1] #alle elementer i motsatt rekkefølge (!)
```

Arrayer støtter vektoriserte operasjoner

```
x = 3.14 * np.ones(1000)
y = n.ones(1000)
```

```
#summere to arrayer komponentvis:
```

```
z = x + y
```

```
#multiplisere alle komponenter med et tall
```

```
z = 2.0 * z
```

```
#multiplisere arrayer komponentvis:
```

```
z = z * x #z[0] = z[0] * x[0], z[1] = z[1] * z[0],...
```

Quiz; hva skrives ut?

```
import numpy as np  
a = [1, 2, 3]  
print(a * 2)  
print(np.array(a) * 2)
```

Hvorfor bruke vektoriserte beregninger?

- Mye raskere beregninger
- Enklere og mer lesbar kode

Eksempel: Regn ut $\sin(x)$ for 11 jevnt fordelte x -verdier i intervallet 0 til 2π .

```
import numpy as np

# Ikke-vektorisert (med lister)
h = 2 * np.pi / 10
xvals = []

for i in range(11):
    xvals.append(sin(i * h))

# Vektorisert (med numpy-arrays)
x = np.linspace(0, 2 * np.pi, 11)
y = np.sin(x)
```

Vektoriserte beregninger i egne funksjoner (1)

```
def ball(t, v0):  
    g = 9.81  
    return v0 * t - 0.5 * g * t**2
```

Hvordan kan denne funksjonen brukes med en array som argument for t (ikke bare tall)?

Vektoriserte beregninger i egne funksjoner (2)

Vektoriserte beregninger krever vanligvis ingen endringer:

- Vanlige matematiske operasjoner ($*$, $+$ etc) støtter arrayer direkte
- Mer avanserte funksjoner må importeres fra Numpy (ikke `math`). `np.sin(x)` fungerer hvis `x` er en array, `math.sin(x)` krever at `x` er et tall

Ett viktig unntak:

- Funksjoner som inneholder if-tester:

```
def heaviside(x):  
    if x < 0:  
        return 0  
    else:  
        return 1
```

Vektoriserte beregninger i egne funksjoner (3)

Hvis `a` er en array vil uttrykk som `a < 5` returnere en array med samme form som `a`:

```
>>> a = np.linspace(0,10,11)
>>> a < 5
array([ True,  True,  True,  True,  True, False, False, False, False,
        False, False])
```

Testen `if a < 5` krever et entydig `True / False`, og gir ikke mening for en array av `True / False` - verdier

Spørsmål for diskusjon

```
def heaviside(x):  
    if x < 0:  
        return 0  
    return 1
```

Hvordan kan vi implementere Heaviside-funksjonen så den støtter arrays?

Mulig løsning: skriv funksjonen på nytt med en for-løkke.

```
def heaviside_loop(x):  
    y = np.zeros_like(x)  
    for i in range(len(x)):  
        y[i] = H(x[i])  
    return y
```

Ulemper:

- Mye å skrive
- Treg kode på grunn av for-løkken

Heaviside for arrays - alternativ 2 (bedre)

Bruk funksjonen `np.vectorize`:

```
def heaviside(x):  
    if x < 0:  
        return 0  
    return 1  
  
heaviside_vec = np.vectorize(heaviside)
```

Fordel:

- Enkel å bruke

Ulempe:

- Like treg som koden med for-løkke

Heaviside for arrays - alternativ 3 (best)

Skriv funksjonen på nytt, med `np.where`:

```
def heaviside_pro(x):  
    return np.where(x < 0, 0.0, 1.0)
```

Fordel:

- Rask kode
- Kort og lesbar kode (for Numpy-eksperter)

Ulemper:

- Må skrive funksjonen på nytt

Del 2: Dictionaries (ordbøker)

En dictionary kan sees som en generalisering av en liste

- Egenskaper ved lister:
 - lagre en *sekvens* av elementer i ett enkelt objekt ([1,3,-1])
 - hvert element er et Python-objekt
 - elementene er indeksert med heltallene 0, 1, ...
- Dictionaries kan indeksere objekter i en samling via tekst eller andre typer nøkler (= «lister med tekstindeks»)
- En dictionary i Python kalles hash, HashMap og assosiativ matrise i andre språk
- Norsk: ordbok, ordliste eller oppslagstabell

Listeindeksen er ikke alltid naturlig

Anta at vi trenger å lagre temperaturene i Oslo, London og Paris.

Løsning med liste:

```
temps = [13, 15.4, 17.5]
cities = ['Oslo', 'London', 'Paris']
print(f'Temperaturen i {cities[0]} er {temps[0]}')
```

- Det ville vært mer naturlig å kode assosiasjonen mellom navnet på byen og temperaturen direkte
- For eksempel skrive `temps["Oslo"]` for å hente ut temperaturen i Oslo

Dictionaries kobler nøkler til objekter

Vi bruker ofte strenger som nøkler, men andre objekter kan også brukes:

```
# Initialiser dictionary
temps = {'Oslo': 13, 'London': 15.4, 'Paris': 17.5}

# Bruk
print('Temperaturen i London er', temps['London'])
print('Temperaturen i Oslo er', temps['Oslo'])
```

Viktig:

- Strengindeksen, som Oslo, kalles en *nøkkel*, mens temps['Oslo'] er den tilhørende *verdien*
- En dictionary er en samling av nøkkel-verdi-par

Initialisering av dictionaries (1)

To måter å initialisere en dictionary på:

```
min_dict = {'nøkkel1': verdi1, 'nøkkel2': verdi2, ...}
```

```
temps = {'Oslo': 13, 'London': 15.4, 'Paris': 17.5}
```

eller

```
min_dict = dict(nøkkel1=verdi1, nøkkel2=verdi2, ...)
```

```
temps = dict(Oslo=13, London=15.4, Paris=17.5)
```

Legg til et nytt element i en dict (dict = dictionary):

```
>>> temps['Madrid'] = 26.0
```

```
>>> print(temps)
```

```
{'Oslo': 13, 'London': 15.4, 'Paris': 17.5,  
'Madrid': 26.0}
```

Initialisering av dictionaries (2)

Vi kan også begynne med en tom dictionary og fylle den med verdier:

```
temps = {}  
temps['Oslo'] = 13  
temps['Stockholm'] = 15.5
```

Eller med en løkke:

```
temps = {}  
for city, temp in zip(cities, temperatures):  
    temps[city] = temp
```

En for-løkke over en dictionary vil iterere gjennom nøklene

```
for nøkkel in dictionary:  
    verdi = dictionary[nøkkel]  
    print(verdi)
```

Eksempel:

```
>>> for by in temps:  
...     print(f'Temperaturen i {by} er {temps[by]}'.  
...  
Temperaturen i Paris er 17.5  
Temperaturen i Oslo er 13  
Temperaturen i London er 15.4  
Temperaturen i Madrid er 26.0
```

Merk: nøklene leses/lagres i samme rekkefølge som de legges inn.

Man kan teste for bestemte nøkler, slette elementer, osv.

Har dict-en en bestemt nøkkel?

```
>>> if 'Berlin' in temps:
...     print('Berlin:', temps['Berlin'])
... else:
...     print('Ingen temperaturdata for Berlin')
...
Ingen temperaturdata for Berlin
>>> 'Oslo' in temps    # standard boolsk uttrykk
True
```

Slett et element fra en dict:

```
>>> del temps['Oslo']    # fjern Oslo-nøkkel med verdi
>>> temps
{'Paris': 17.5, 'London': 17.5, 'Madrid': 26.0}
>>> len(temps)          # antall nøkkel-verdi-par i dict.
3
```

Nøklene og verdiene kan hentes ut separat

Metodene `temps.keys()` og `temps.values()` returnerer nøklene og verdiene som *iteratorer*, som er listelignende objekter man kan løpe gjennom:

```
>>> for by in temps.keys():
>>>     print(by)
...
Paris
Madrid
London
>>> for t in temps.values():
>>>     print(t)
17.5
26.0
17.5
```

Mer nyttig: hent ut nøkler og verdier samtidig

Metoden `temps.items()` henter ut nøkler og verdier som tupler

```
>>> for par in temps.items():  
>>>     print(par)  
('Paris', 17.5)  
('Madrid', 26.0)  
('London', 17.5)
```

Forsiktig: to variabler kan endre den samme dictionaryen

```
>>> t1 = temps
>>> t1['Stockholm'] = 10.0      # endre t1
>>> temps                       # temps er også endret!
{'Stockholm': 10.0, 'Paris': 17.5, 'London': 15.4,
 'Madrid': 26.0}
>>> t2 = temps.copy()          # ta en kopi
>>> t2['Paris'] = 16
>>> t1['Paris']                 # t1 ble ikke endret
17.5
```

Det samme gjelder for lister:

```
>>> L = [1, 2, 3]
>>> M = L
>>> M[1] = 8
>>> L[1]
8
>>> M = L[:]   # ta kopi av L
>>> M[2] = 0
>>> L[2]
3
```

Nøklene i en dictionary må være 'immutable'

- Hittil: nøkkelen er tekst (streng-objekt)
- Nøkler kan være ethvert *immutable* (konstant) objekt (!)

Immutable:

- int-verdier og float-verdier
- string-verdier
- tupler

Ikke immutable:

- lister (kan endre element med `a[i]=y`)
- objekter av egendefinerte klasser

Eksempler på nøkler

```
>>> d = {1: 34, 2: 67, 3: 0}           # nøkkel er int
>>> d = {13: 'Oslo', 15.4: 'London'}  # mulig
>>> d = {(0,0): 4, (1,-1): 5}         # nøkkel er tuppel
>>> d = {[0,0]: 4, [-1,1]: 5}         # liste er mutabel/endringsbar
...
TypeError: unhashable type: 'list'
```

- To elementer kan ikke ha samme nøkkel: `<linebreak>` Vi kan skrive `d = {'Oslo': 10.5, 'Oslo': 15}`, men det blir bare en nøkkel i dictionaryen, siden den siste vil overskrive den første
- To elementer kan ha samme verdi: `<linebreak>` Vi kan godt skrive `d = {'Oslo': 10.5, 'Bergen': 10.5}`.
- En dictionary lagrer nøkkel-verdi-par i samme rekkefølge som de ble satt inn (Python 3.7 eller nyere).

Nøstede dictionaries

- Verdiene i en dictionary kan være hva som helst, også nye dictionaries
- Akkurat som med lister kan vi ha en *dictionary av dictionaries*, som kalles en *nøstet dictionary*

Eksempel:

```
grunnstoffer = {"H": {"masse": 1.01, "elektronegativitet": 2.2,
                      "fase": "gass"},
                 "V": {"masse": 50.94, "elektronegativitet": 1.6,
                      "fase": "fast"}}

print(grunnstoffer["H"]["fase"])
print(grunnstoffer["V"]["elektronegativitet"])
```

Eksempel: Polynom representert av dictionaries

Informasjonen i polynomet

$$p(x) = -1 + x^2 + 3x^7$$

kan representeres av en dict med potens som nøkkel (int) og koeffisient som verdi (float):

```
p = {0: -1, 2: 1, 7: 3.5}
```

Evaluer et slikt polynom $\sum_{i \in I} c_i x^i$ for en gitt verdi x :

```
def eval_poly_dict(poly, x):  
    sum = 0.0  
    for potens in poly:  
        sum += poly[potens]*x**potens  
    return sum
```

Kortere proffversjon:

```
def eval_poly_dict(poly, x):  
    # Python's sum kan addere elementer i en iterator  
    return sum(poly[potens]*x**potens for potens in poly)
```

Polynomer kan også representeres av lister

Listeindeksen tilsvarer potensen, f.eks. dataene i $-1 + x^2 + 3x^7$ representeres som

```
p = [-1, 0, 1, 0, 0, 0, 0, 3]
```

Det generelle polynomet $\sum_{i=0}^N c_i x^i$ lagres som $[c_0, c_1, c_2, \dots, c_N]$.

Evaluer et slikt polynom $\sum_{i=0}^N c_i x^i$ for en gitt x :

```
def eval_poly_list(poly, x):  
    sum = 0  
    for potens in range(len(poly)):  
        sum += poly[potens]*x**potens  
    return sum
```

Hva er best for polynomer: lister eller dictionaries?

Dictionaries trenger bare å lagre leddene som ikke er null.

Sammenlign dict vs. liste for polynomet $1 - x^{200}$:

```
p = {0: 1, 200: -1}          # len(p) er 2  
p = [1, 0, 0, 0, ..., 200]  # len(p) er 201
```

Dictionaries kan enkelt håndtere negative potenser, f.eks.,

$$\frac{1}{2}x^{-3} + 2x^4$$

```
p = {-3: 0.5, 4: 2}  
print(eval_poly_dict(p, x=4))
```