

5 List, og for løkka

Læringsmål dette kapittel:

- Forstå behovet for lister når flere dataelementer skal behandles
- Kjenne til `list` som en innebygd datatype i Python
- Forstå `list` sine egenskaper som en container
- Forstå `list` sine egenskaper som en sequence
- Kunne opprette lister (tomme, med verdier, med blandet innhold) ved hjelp av literaler eller konstruktøren
- Forstå at lister er mutable (foranderlige) og kunne endre elementer ved hjelp av indeksering
- Kunne skille mellom kopiering av en liste-referanse og en faktisk kopi av en liste
- Kunne utføre shallow copy (overfladisk kopi) av lister ved hjelp av ulike metoder
- Forstå implikasjonene av shallow copy, spesielt når listen inneholder mutable objekter
- Kunne utføre deep copy (dyp kopi) av lister ved hjelp av `copy.deepcopy()` for fullstendig uavhengighet av originalen
- Kunne bruke indeksering (positive og negative) for å hente ut eller endre enkelt-elementer i en liste
- Kunne bruke avansert slicing for å hente ut delsekvenser fra lister
- Forstå hvordan et slicing-uttrykk kan brukes på venstre side av en tilordning for å erstatte, fjerne eller sette inn elementer i en liste
- Kjenne til og kunne bruke vanlige operasjoner på sekvenser som `len()`, `min()`, `max()`, `sum()`, `sorted()`, `reversed()`, `enumerate()`, og `zip()`
- Kjenne til og kunne bruke list klassens medlemsfunksjoner som `append()`, `insert()`, `remove()`, `pop()`, `index()`, `count()`, `sort()`, `reverse()`, og `clear()`
- Vite at metoden `sort()` sorterer listen "in-place" og returnerer `None`.
- Forstå og kunne bruke `for`-løkken for å iterere gjennom elementer i en liste (eller annen sekvens)

- Forstå skillet mellom "ikke-destruktiv" (iterere over en kopi) og "destruktiv" (endre originalen) iterasjon med for-løkken, og når man trenger indeks for å endre listen
- Kjenne til konvensjonen med _ (understrek) som en "anonym" iterasjonsvariabel når verdien ikke skal brukes
- Kunne bruke for-løkker for å lage en kopi av en liste
- Kunne bruke range() og enumerate() sammen med for-løkken for å få tilgang til indeks og verdier under iterasjon
- Forstå hvorfor str datatypen er immutable i kontrast til list
- Kunne opprette og iterere gjennom flerdimensjonale lister (inkludert "ragged lists")
- Kjenne til muligheten for generisk iterering av ukjent antall dimensjoner (f.eks. ved rekursjon)
- Kjenne til og kunne bruke list comprehensions

Mange lærebøker introduserer *løkkestrukturer*, det vil si **for** – løkker og **while** løkker før datastrukturen **list** presenteres.

Det er mye å si om for – løkker, så vi kommer tilbake til det i et eget kapittel, men lister er nesten umulig å nevne uten også å snakke om **for** – løkker; - vi «må» ha en løkke for å gjøre noe «fornuftig» med en liste.

Så, **list** - datastrukturen og **for** – løkkestrukturen er som hånd og hanske; - de hører sammen.

5.1 Hvorfor trenger vi lister?

Hittil har vi brukt individuelle / enkeltstående variabler i programmene våre, dvs hver variabel kan kun inneholde *en* verdi. Så snart vi har behov for å behandle flere dataelementer, gjerne av samme type, får vi bruk for mer avanserte datastrukturer.

Tenk for eksempel om vi fikk i oppgave å lese inn 10 heltall, summere og sortere disse og skrive ut resultatet.

En naiv tilnærming ville være å lage 10 forskjellige heltallsvariabler, og lese inn verdi til disse fra bruker:

```
int1 = int(input("First number:"))
int2 = int(input("Second number:"))
int3 = int(input("Third number:"))
# etc..
# alle int1 til int10 er lest, men hva nu?
```

Det finnes ingen elegant måte å summere 10 individuelle heltall på, det ville blitt omtrent slik:

```
sum = int1 + int2 + int3 + ... + int10
```

Sortering, enda verre; - greit nok å sortere 3 individuelle tall, men hva med 10, hvordan ville `if - else` konstruksjonen se ut? 😊

Mildt sagt håpløst..., hva om vi skulle lese 100 tall?

Her kommer ***lister*** og ***for løkker*** inn i bildet. Vi kan nå løse oppgaven elegant:

```
1: # file: sc_05.01.py
2: # les inn 10 heltall til en liste, summerer og sorter
3:
4: heltall_liste = []
5: for i in range(10):
6:     tall = int(input(f"Skriv inn heltall nr {i + 1} av 10:"))
7:     heltall_liste.append(tall)
8:
9: summen = sum(heltall_liste)
10: print("Summen av de innleste tallene er:", summen)
11: heltall_liste.sort()
12: print("Tallene, sortert:", heltall_liste)
```

Programmet vil oppføre seg slik:

```
Skriv inn heltall nr 1 av 10: 2
Skriv inn heltall nr 2 av 10: 4
Skriv inn heltall nr 3 av 10: 5
Skriv inn heltall nr 4 av 10: 6
Skriv inn heltall nr 5 av 10: 71
Skriv inn heltall nr 6 av 10: 0
Skriv inn heltall nr 7 av 10: 87
Skriv inn heltall nr 8 av 10: 45
Skriv inn heltall nr 9 av 10: 67
Skriv inn heltall nr 10 av 10: 78
Summen av de innleste tallene er: 365
Tallene, sortert: [0, 2, 4, 5, 6, 45, 67, 71, 78, 87]
```

Linje 4

Her oppretter vi en tom liste. Python vet at det er en liste fordi vi bruker `[]` parantesene. Lista er tom fordi det ikke er noe inni parentesene.

Linje 5

Dette er en **for** løkke som itererer 10 ganger ved hjelp av **in range** konstruksjonen, legg merke til avslutning av linja med **kolon**. **in range** er bare en av flere måter å få en for løkke til å iterere et bestemt antall ganger.

i er en *iterasjonsvariabel* som løper fra 0 opp til og med 9, altså en mindre en argumentet til **in range** konstruksjonen. Legg merke til at vi bygger opp en tekst til brukeren som utnytter denne til å si hvilket tall vi er i ferd med å lese inn 😊

Linje 6 og 7

En blokk med kode som utføres for hver iterasjon av for løkka; - det er viktig at disse linjene har samme innrykk; - da tilhører de samme blokk.

append(tall) sørger for å legge et nytt heltall til **heltall_liste**, som vokser med ett og ett element for hver iterasjon av lista.

Linje 9

Her bruker vi den Python-innebygde funksjonen **sum()** til å produsere summen av heltallene i lista.

Linje 11

Her bruker vi klassen **list** sin medlemsfunksjon **sort()** til å sortere lista. Sorteringen skjer **in place**, dvs *selve lista sorteres, det skapes ikke en sortert kopi*. En vanlig misforståelse er at **sort()** returnerer noe (man tror, ei sortert liste); - det stemmer ikke; - «returverdien» fra **sort()** er **None**.

Linje 12

En **print** setning som har en ledetekst "Tallene, sortert:", pluss at en oppgir variabelnavnet **heltall_liste** i **print()** funksjonskallet.

Det å bruke **heltall_liste** som argument til **print()** funksjonen fører altså til at flg skrives ut:

```
[0, 2, 4, 5, 6, 45, 67, 71, 78, 87]
```

Hvordan vet **print()** funksjonen hvordan den skal skrive ut en liste?

Bak kulissene

Når **print()** funksjonen har en variabel som argument, f eks **heltall_liste**, så ser den etter, og kaller en medlemsfunksjon som

heter `__str__()` i klassen til variabelen, her klassen **list**. Det er denne medlemsfunksjonen som returnerer en *streng formattert som ønsket* for aktuell datatype (klasse). Python programmerne har bestemt at en **list** sin `__str__()` skal returnere en streng som formatteres slik:

`[0, 2, 4, 5, 6, 45, 67, 71, 78, 87]`

... hvilket er fornuftig, en ser umiddelbart at dette er en *liste* på grunn av parentesene [og].

5.2 List er en innebygd datatype

I Python er **list** en *innebygd datatype* (engelsk: *built-in data type*) en datatype som er direkte tilgjengelig i språket uten at du trenger å importere noe. Disse datatypene er en del av kjernen i Python og gir grunnleggende funksjonalitet for å representere og manipulere en samling av data.

Disse andre er **tuple**, **set**, **dict** og **frozenset**, vi omtaler gjerne disse som container-klasser, eller samlingsklasser.

Python har også en egen modul kalt **collections** som inneholder mer avanserte samlingstyper, disse er:

namedtuple, **deque**, **Counter**, **OrderedDict**, **defaultdict**, **ChainMap**.

Vi kommer tilbake til de andre innebygde container klassene i detalj, og enkelte av klassene i **collections** modulen

5.3 List er en container

Container er et begrep som brukes i Python (og andre språk), men det er ikke en formell *type* i språket slik som **list**, **dict**, **set** og **tuple**. Det er mer et beskrivende eller konseptuelt begrep som refererer til datastrukturer *som inneholder flere elementer*.

Ikke alle containere er en sequence (se under).

5.4 List er en sequence

I Python er en *sequence* (sekvens) en *ordnet samling* av *elementer* der hvert element har en *indeks* (posisjon), og man kan *iterere* over elementene i rekkefølge. Dette er en grunnleggende og viktig kategori i Python fordi den gir et felles sett med egenskaper og operasjoner som gjelder for flere ulike datatyper.

En **sequence** er et objekt som:

- Har en **definert rekkefølge** på elementene.
- Støtter **indeksering** (f.eks. `s[0]` for første element).
- Støtter **slicing** (f.eks. `s[1:4]`).
- Kan **itereres** over i en **for-løkke**.
- Har en **lengde** (`len(s)`).
- Støtter ofte operasjoner som `in`, `+`, `*`, og sammenligning.

`set` og `dict` som vi skal lære om senere er ikke sequences; - `set` fordi den ikke er ordnet og heller ikke indekserbar, `dict` fordi den ikke er indekserbar.

Av de innebygde datatypene er `list`, `tuple`, `str` og `range` *sequences*.

5.5 Opprette en liste, legge til elementer

Koden under viser ulike (enkle) måter å opprette en liste på.

```

1: # file: sc_05_02.py
2:
3:
4: # Initialisere en tom liste
5: number_list = []
6: # initialisere en tom liste med constructor
7: number_list2 = list()
8:
9: # Initialisere en liste med noen verdier
10: name_list = ["anna", "bjorn", "clara"]
11: # Initialisere en liste med verdier med
constructor
12: name_list2 = list(["john", "paul", "liam"])
13:
14: # Initialisere en liste med blandet innhold
15: mixed_list = [1, "tekst", 3.14, True]
16:

```

```

17: # Legge til elementer i listen
18: number_list.append(1)
19: number_list.append(2)
20: number_list2.append(10)
21:
22: # Endre elementer i en liste med indeksering
23: name_list[0] = "suzanne"      # Endrer første
element
24: mixed_list[1] = "ny_tekst"    # Endrer andre
element
25:
26: # Skrive ut listene
27: print(number_list)    # [1, 2]
28: print(number_list2)   # [10]
29: print(name_list2)     # ['john', 'paul', 'liam']
30: print(name_list)      # ['suzanne', 'bjorn',
'clara']
31: print(mixed_list)     # [1, 'ny_tekst', 3.14, True]
32:
33: # initialisere fra en streng
34: tekst = "python"
35: bokstaver = list(tekst)
36: print(bokstaver)       # ['p', 'y', 't', 'h', 'o',
'n']
37: ord = [tekst]
38: print(ord)            # ['python']

```

Linje 5

En *tom* liste opprettes enklest slik.

Parentes-typen [] forteller at vi ønsker å opprette en *liste*.

Slike parenteser: (), forteller at vi ønsker å opprette en **tuple**, og disse {} at vi ønsker å opprette en **dict**. Smart, ikke sant?

Linje 7

Her opprettes en *tom* liste ved hjelp av **list** konstruktøren (list constructor). Denne koden gjør det samme som koden i linje 5.

List constructor

Alle klasser har en metode som er det samme som navnet på klassen, altså **list** sin er **list()**. Dette kalles klassens *konstruktør*; - den

brukes til å initialisere objektet som skapes med en ønsket verdi. Se eksempler i koden.

Linje 10 og 12

Bruker initialisering med *literaler* (linje 10) og constructor for å lage lister med innhold, denne gangen tekst. Legg merke til [] parantesene i linje 12. Hadde ikke de vært der ville du fått en feilmelding om at list constructoren kun aksepterer *en* såkalt iterable, ikke flere. Mer om iterabler senere.

Literal?

En *literal* er en *fast verdi* som er skrevet direkte i programkoden. Det er en konkret representasjon av en verdi som Python forstår uten at det kreves noen beregning, funksjonskall eller variabel.

Iterable?

En *iterable* i Python er et objekt som kan "itereres over" – det vil si at du kan gå gjennom elementene ett for ett, vanligvis i en løkke som forløkke. Dette er en grunnleggende del av hvordan Python håndterer sekvenser og datastrukturer. Mye mer om iterable senere, på ulike steder.

Linje 15

En liste kan inneholde elementer av *forskjellig type*.

Linje 18 - 20

Vi bruker metoden **append()** for å legge til elementer **til slutten** av en liste. Når vi har et objekt som f eks **number_list**, som er av typen **list** (klassen **list**), så kaller vi en metode ved hjelp av *dot-notasjonen*, altså **number_list.append()**.

Linje 23 - 24

Lister er *mutable*, det vil si vi *kan endre innholdet*. Vi husker at strenger, **str** datatypen *ikke* kunne endres, altså vi kan ikke skrive følgende:

```
>>> navn = "john"
>>> navn[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: 'str' object does not support item assignment
```

```
>>>
```

Vi ser at for lister går det fint å endre et element ved å bruke *indeksering* til å «skrive inn i» en lokasjon:

```
name_list[0] = "suzanne"      # Endrer første element
```

Linje 34 - 36

Vær obs på denne: Her brukes list constructoren til å ta imot en *streng*. List constructoren aksepterer **ett** itererbart (iterable) objekt, og tekster er et slikt, nemlig et **str** objekt. Når constructoren itererer over tekster, så henter den ut *ett og ett element*, så derfor blir resultatet en *liste av tegn* (som egentlig er str objekter...), ikke ett ord.

For å få laget ei liste med bare en streng, så må vi enten skrive (gitt initialiseringen `tekst = "python"`)

```
ord = list([tekst])
```

eller

```
ord = [tekst]
```

5.6 indeksering og lister

Lister i Python er *indekserte datastrukturer*, noe som betyr at hvert element har en posisjon (indeks) som kan brukes for å hente eller endre verdier:

`liste[indeks]`

`liste` er navnet på listen, `indeks` er et heltall som angir posisjonen til elementet (starter på 0)

```
navn = ["Anna", "John", "Clara"]
print(navn[0])  # Skriver ut: Anna
```

Her brukes `navn[0]` til å hente det første elementet i listen.

Når indeksering brukes *på høyre side*, betyr det at vi *henter* en verdi fra listen:

```
first = navn[0]
```

Her kopieres verdien på indeks 0 til variabelen `first`, listen endres ikke.

Når indeksering brukes *på venstre side*, betyr det at vi *endrer* verdien i listen på den gitte posisjonen:

```
navn[1] = "Beatrice"
```

Nå erstattes "John" med "Beatrice" i listen, listen blir: ["Anna", "Beatrice", "Clara"]

5.6.1 Negative indekser

Du kan bruke *negative indekser* for å telle fra *slutten*:

```
last = navn[-1]
```

gir *siste* element, `last` får verdien "Clara"

5.6.2 Ulovlige indekser

Du får en `IndexError` dersom du bruker en indeks som er utenfor lovlig intervall, altså mindre enn null og større enn lengden av lista minus en.

En `IndexError` vil føre til at programmet krasjer. Vi kan bruke *exception handling* for å hindre dette.

5.7 Operasjoner på lister

Vi så i kodeksempelet over at vi brukte den globale funksjonen `sum()` på lista. Det er en rekke funksjoner og operatorer som vi kan bruke på sequences. Siden `list` er en slik, kan vi altså bruke alle systemfunksjoner og operatorer i tabellen under på en liste:

Forutsatt

```
seq1 = [1, 2, 3] # altså en liste
seq2 = [4, 5, 6] # også en liste
```

Så gjelder følgende:

Operasjon	Beskrivelse	Eksempel
-----------	-------------	----------

<code>len(seq1)</code>	Antall elementer i sekvensen	<code>len(seq1) → 3</code>
<code>seq1[i]</code>	Hente element på indeks `i`	<code>seq1[1] → 2</code>
<code>seq1[i:j]</code>	Slicing – delsekvens fra `i` til `j-1`	<code>seq1[0:2] → [1, 2]</code>
<code>seq1 + seq2</code>	Konkatinering (sammenslåing)	<code>seq1 + seq2 → [1, 2, 3, 4, 5, 6]</code>
<code>seq1 * 2</code>	Multiplikasjon	<code>seq1 * 2 → [1, 2, 3, 1, 2, 3]</code>
<code>x in seq1</code>	Sjekk om `x` finnes i sekvensen	<code>if 2 in seq1 → True</code>
<code>min(seq2) / max(seq2)</code>	Minimum / maksimum verdi	<code>min(seq2) → 4 max(seq2) → 6</code>
<code>sum(seq1)</code>	Summerer tall i sekvensen	<code>sum(seq1) → 6</code>
<code>sorted(seq2)</code>	Returnerer sortert <i>kopi</i> (i motsetning til <code>list</code> medlemsmetoden <code>sort()</code> , som sorterer in-place og returnerer <code>None</code>)	<code>sorted(seq2) → [4, 5, 6]</code>
<code>reversed(seq1)</code>	Returnerer reversert iterator	<code>list(reversed(seq1)) → [3, 2, 1]</code>
<code>range(start, stop[, step])</code>	Genererer en tallsekvens	<code>range(1, 5) → [1, 2, 3, 4]</code>
<code>enumerate(seq1)</code>	Returnerer (indeks, verdi)-par	<code>list(enumerate(seq1)) → [(0, 1), (1, 2), (2, 3)]</code>
<code>zip(seq1, seq2)</code>	Slår sammen elementer fra to sekvenser	<code>list(zip(seq1, seq2)) → [(1, 4), (2, 5), (3, 6)]</code>
<code>seq1 == [1, 2, 3]</code>	Sammenligning av sekvenser	True

seq1 < seq2	Leksikografisk sammenligning	True (fordi 1 < 4)
-------------	------------------------------	--------------------

5.8 Klassen `list` sine medlemsfunksjoner

Forrige kapittel fokuserte på operasjoner på lister, men som også var *generelle* for sequences.

`list` klassen har mange medlemsmetoder som opererer på lista, her følger noen av de viktigste. Vi tar utgangspunkt i at vi har en *shopping_list* og ser hva vi kan gjøre med den:

```
shopping_list = ['milk', 'bread', 'cheese']
```

append() – Legg til et element på slutten

```
shopping_list.append('butter')
print(shopping_list) # ['milk', 'bread', 'cheese',
'butter']
```

insert() – Sett inn et element på en bestemt plass

```
shopping_list.insert(1, 'egg')
print(shopping_list) # ['milk', 'egg', 'bread',
'cheese', 'butter']
```

remove() – Fjern første forekomst av et element

```
shopping_list.remove('bread')
print(shopping_list) # ['milk', 'egg', 'cheese',
'butter']
```

pop() – Fjern og returner et element (*siste* som *standard*)

```
sist = shopping_list.pop()
print(sist) # 'butter'
print(shopping_list) # ['milk', 'egg', 'cheese']
```

index() – Finn posisjonen til et element

```
posisjon = shopping_list.index('egg')
print(posisjon) # 1
```

count() – Tell hvor mange ganger et element *forekommer*

```
shopping_list.append('milk')
antall = shopping_list.count('milk')
print(antall) # 2
```

sort() – Sorter listen i stigende rekkefølge

```
shopping_list.sort()
print(shopping_list) # ['egg', 'milk', 'milk',
'cheese']
```

reverse() – Snu rekkefølgen i listen

```
shopping_list.reverse()
print(shopping_list) # ['cheese', 'milk', 'milk',
'egg']
```

extend() – Utvid listen med en iterable (f eks list, string, tuple)

```
shopping_list.extend(['apple', 'juice'])
print(shopping_list) # ['cheese', 'milk', 'milk',
'egg', 'apple', 'juice']
```

clear() – Tøm hele listen

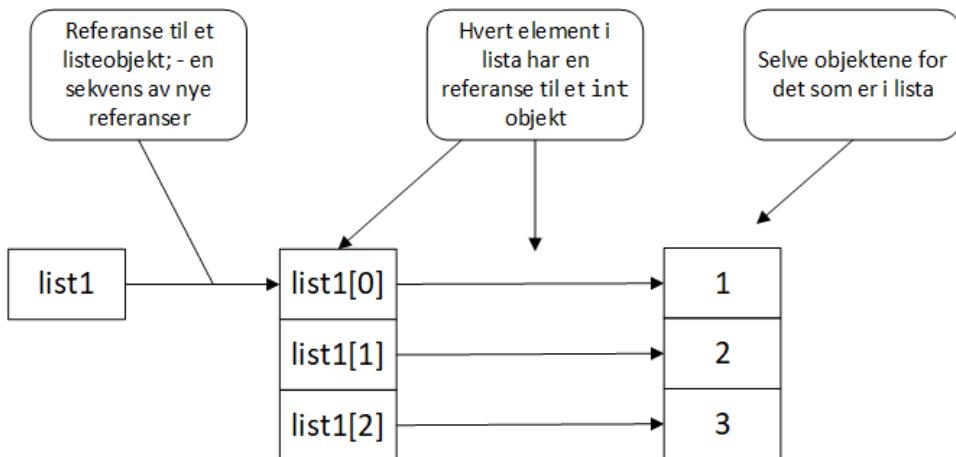
```
shopping_list.clear()
print(shopping_list) # []
```

5.9 En liste er en liste av referanser

I Python er alle variabler egentlig referanser til objekter i minnet. Når vi skriver `a = 5`, betyr det at navnet `a` peker til et objekt av typen `int` med verdien 5. Dette gjelder for alle typer – også lister.

En liste er et objekt som inneholder en *sekvens av referanser* til andre objekter. Når vi skriver `list1 = [1, 2, 3]`, peker `list1` til et listeobjekt, og dette objektet inneholder referanser til tre `int`-objekter med verdiene 1, 2 og 3.

Ovennevnte kan illustreres slik:



Figur 5.1 Intern organisering av en liste

5.9.1 Kopiere lister

5.9.1.1 Kopiering av referansen til lista er IKKE kopiering!

Dette er en kopiering av en *referanse*:

`list2 = list1`

Det er **ikke** en kopiering av selve lista. Tilordningen over fører til at referansen `list2` nå refererer / peker til det samme som referansen `list1`. Dersom `list2` refererte til en annen liste før tilordningen, så vil det innholdet bli tapt.

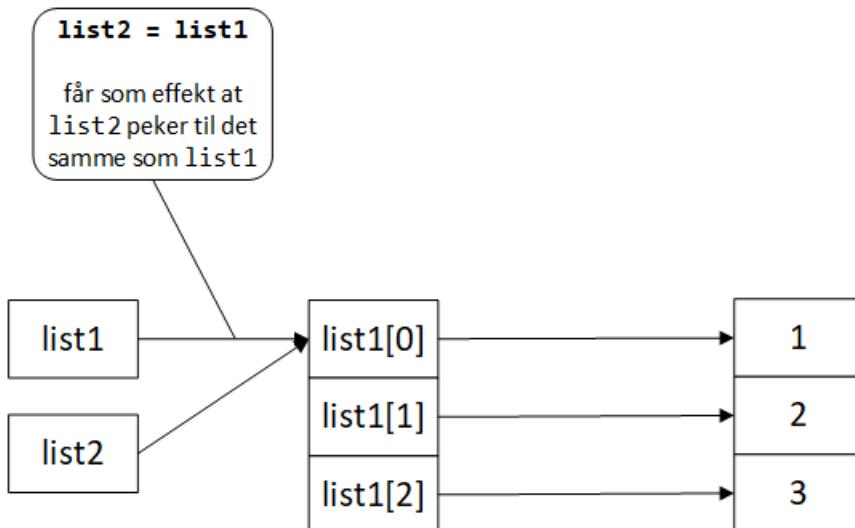
Se på følgende kodesnutt utført i REPL:

```

>>> list1 = [1,2,3]
>>> id(list1)
2692026799872
>>> list2 = list1
>>> id(list2)
2692026799872
>>>

```

Vi ser at `id` er den samme for referansene `list1` og `list2`. Dette er illustrert i Figur 5.2



Figur 5.2 Ingen kopiering, kun listereferansen kopieres

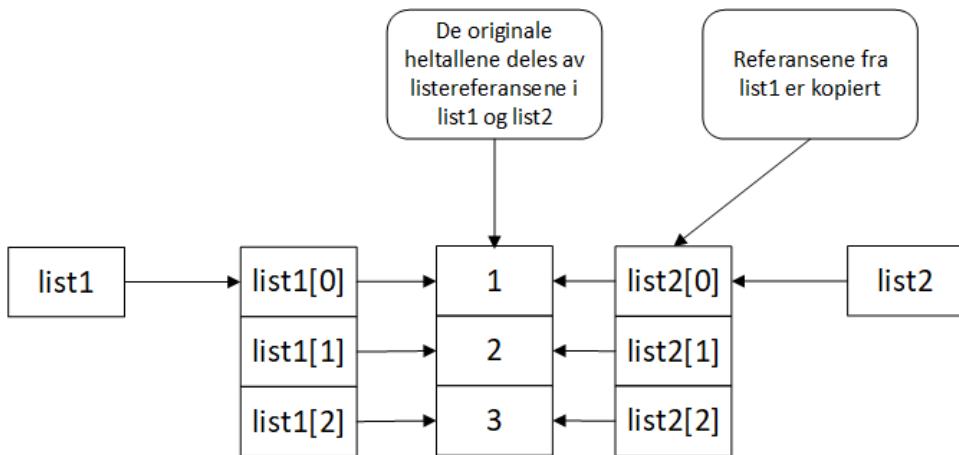
5.9.1.2 Kopiering: shallow copy

Merk at lister også kan kopieres ved hjelp av en for - løkke og bruk av append(), eller flere for løkker hvis lista er flerdimensjonal, lær om dette under for løkker nedenfor.

Generelt om shallow copy:

En *shallow copy* (overfladisk kopi) er en kopi av en samling (for eksempel en liste) hvor selve samlingen kopieres, men ikke objektene den inneholder. Det betyr at den nye samlingen får referanser til de samme objektene som originalen. Endrer du et *immutable* element i kopien (for eksempel bytter ut et tall), påvirker det ikke originalen. Men hvis elementene er *mutable* objekter (for eksempel lister i en liste, altså flerdimensjonale lister), vil endringer i disse objektene være synlige i begge kopier.

Etter at vi har gjort en shallow copy (her med immutable objekter av `int` typen) har vi denne situasjonen:



Figur 5.3 Resultat etter shallow copy

Det er flere måter å oppnå en shallow copy på, som vist i koden under; - her er også vist kun kopiering av referanser, samt bruk av list klassen sin `copy()` metode, samt `copy` modulen sine `copy()` og `deepcopy()` metoder:

```

01: # file : sc_05_04.py
02: import copy
03:
04: list1 = [1, 2, 3]
05:
06: # Bare referanse kopiering (ingen kopi)
07: list2 = list1
08: list2[0] = 10
09: print("Etter referanse-kopiering:")
10: print("list1:", list1) # [10, 2, 3]
11: print("list2:", list2) # [10, 2, 3]
12:
13: # Shallow copy med slicing
14: list1 = [1, 2, 3]
15: list2 = list1[:]
16: list2[0] = 30
17: print("\nEtter shallow copy med slicing:")
18: print("list1:", list1) # [1, 2, 3]
19: print("list2:", list2) # [30, 2, 3]
20:
21: # Shallow copy med list comprehension

```

```
22: list1 = [1, 2, 3]
23: list2 = [x for x in list1]
24: list2[0] = 40
25: print("\nEtter shallow copy med comprehension:")
26: print("list1:", list1) # [1, 2, 3]
27: print("list2:", list2) # [40, 2, 3]
28:
29: # Shallow copy med +
30: list2 = [] + list1
31: list2[0] = 60
32: print("\nEtter shallow copy med +:")
33: print("list1:", list1) # [1, 2, 3]
34: print("list2:", list2) # [60, 2, 3]
35:
36: # Shallow copy med list klassen sin copy() metode
37: list1 = [1, 2, 3]
38: list2 = list1.copy()
39: list2[0] = 20
40: print("\nEtter shallow copy med copy():")
41: print("list1:", list1) # [1, 2, 3]
42: print("list2:", list2) # [20, 2, 3]
43:
44: # Shallow copy med copy.copy()
45: list1 = [1, 2, 3]
46: list2 = copy.copy(list1)
47: list2[0] = 30
48: print("\nEtter shallow copy med copy.copy():")
49: print("list1:", list1) # [1, 2, 3]
50: print("list2:", list2) # [30, 2, 3]
51:
52: # Deep copy
53: list1 = [1, 2, 3]
54: list2 = copy.deepcopy(list1)
55: list2[0] = 50
56: print("\nEtter deep copy:")
57: print("list1:", list1) # [1, 2, 3]
58: print("list2:", list2) # [50, 2, 3]
```

Se figur nedenfor som illustrerer hva som har skjedd i forbindelse med en shallow copy (som i koden her) hvor lista inneholder immutable elementer (som her, heltall, **int** datatypen).

Linje 6 – 11

Illustrerer det som altså *ikke* er en kopiering av lister, kun en kopiering av *referansen* til en liste; - så `list1` og `list2` peker til samme liste. Se Figur 5.2 Ingen kopiering, kun listereferansen kopieres

Linje 13 – 19

Viser shallow kopiering ved hjelp av slicing. Linje 14:

```
list2 = list1[:]
```

Når både `start` og `end` i slicing syntaksen er utelatt, betyr det at `start` settes til 0, og `end` settes til lengden av lista minus `en`, `len(list1)`. Så hele lista kopieres. Situasjonen er nå som i Figur 5.3. I linje 16 endrer vi heltallet som refereres av `list2[0]`:

```
list2[0] = 30
```

Effekten av dette kan synes overraskende, se linje 18 og 19 og Figur 5.4, men husk at `int` datatypen er *immutable*, så vi får samme effekt som vi lærte om i kapittel 2, se Figur 2.2.

Linje 21 – 27

Shallow copy utført med en såkalt *list comprehension*. En comprehension er rett og slett en komprimert utgave av en `for` løkke, vi kommer tilbake til denne mekanismen senere. Effekten er som forrige kodelinjer.

Linje 29 – 34

Shallow copy utført med addisjon til en tom liste.

Linje 36 – 42

Shallow copy utført med `list` klassen sin egen `copy()` metode. Dette er den foretrukne måten å kopiere en liste dersom du ønsker en shallow copy.

Linje 44 – 50

Shallow copy utført med `copy()` metoden fra `copy` modulen. Se at `copy` modulen er importert i linje 1.

Linje 52 – 58

Se neste delkapittel om deep copy

Når bør du ikke bruke shallow copy?

Når listen inneholder **mutable objekter** som du planlegger å endre.
Når du jobber med **flerdimensjonale lister** og vil unngå at endringer i kopien påvirker originalen.

Når du ikke har full kontroll over hvordan kopien skal brukes senere.

Dypdykk: Kopiering av lister i Python

Når du skal lage en *shallow copy* av en liste i Python, er den innebygde metoden `list.copy()` den anbefalte og mest direkte måten å gjøre det på:

```
min_liste = [1, 2, 3]
kopi = min_liste.copy()
```

Denne metoden er:

- Rask – fordi den er implementert direkte i `list` - klassen.
- Trygg – gir en ny liste med referanser til de samme elementene.

Hva med `copy.copy()`?

Modulen `copy` inneholder en funksjon med samme navn:

`copy.copy()`. Den er mer generell og brukes når du ikke vet hvilken type objekt du kopierer.

```
import copy
kopi = copy.copy(min_liste)
```

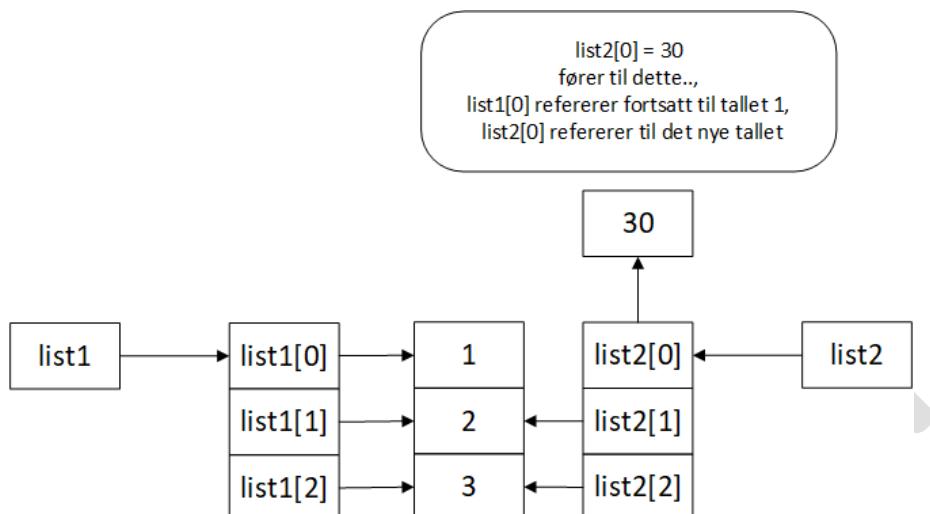
Bak kulissene gjør `copy.copy()` følgende:

1. Sjekker om objektet har en optimalisert `copy()` metode for kopiering (som `list.copy()`). En slik finnes i noen av de innebygde lagringsklassene i Python.
2. Hvis ikke, ser den etter en `__copy__()` - metode i objektets klasse.
3. Hvis ingen finnes, forsøker den å lage en shallow copy basert på objektets metadata. Metadata er «data om data», og Python kan spørre etter slikt.

Anbefaling

- Bruk `list.copy()` når du jobber med lister – det er den mest effektive og idiomatiske måten.

- Bruk `copy . copy()` når du jobber med objekter av ukjent type, eller når du skriver generisk kode.
- Bruk `copy . deepcopy()` når du trenger en fullstendig kopi av alle nivåer i en datastruktur.

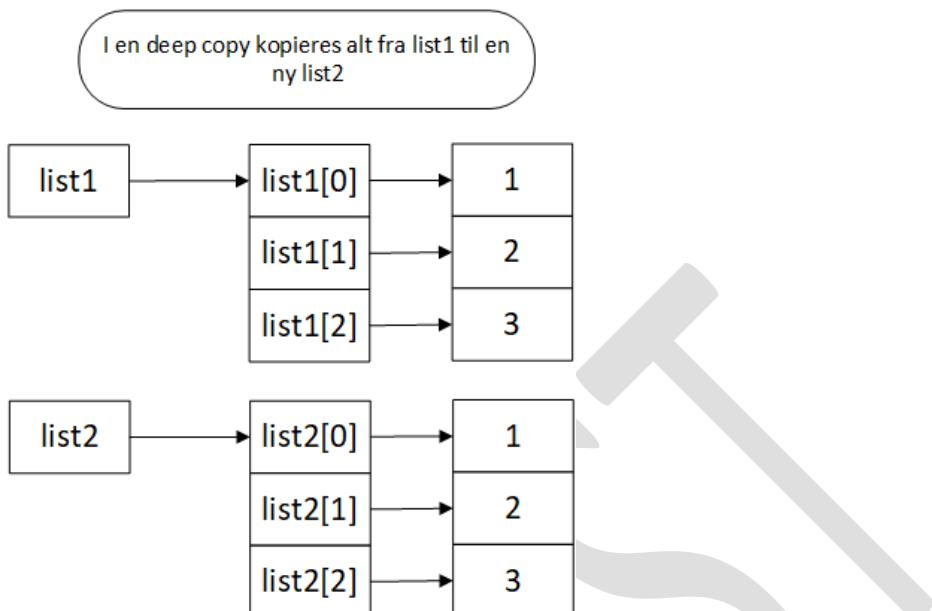


Figur 5.4 Shallow copy; - endring av et element referert fra list2

5.9.1.3 Kopiering: deep copy

I linje 52 – 58 utfører vi en **deep copy**.

Det er en kopiering som gjør en *fullstendig kopi* av innholdet i lista, så innholdet i lista, `int` elementene, blir også kopiert. Se Figur 5.5



Figur 5.5 Deep copy - alt kopieres

5.10 for løkka – *selve motoren*

for - løkka er en grunnleggende mekanisme for å løpe gjennom hvert enkelt element i ei liste og andre containerklasser som er *sequences*.

Gitt at vi har en liste numbers:

```
numbers = [3, 7, 2, 8, 4]
```

Ei liste som den over, samt andre sequences, er en såkalt *itererbar* datastruktur, det vil si vi kan *løpe gjennom den*, eller *iterere* over den. La oss generelt kalle den for en **iterable** i teksten nedenfor.

Her er tre ulike teknikker for å løpe gjennom en iterable:

Teknikk 1:

```
for element in iterable:
```

Syntaks:

```
for element in iterable:  
    # gjør noe med element
```

Forklaring:

- Itererer direkte over hvert element i iterable
- element er en kopi av verdien, ikke en referanse til plasseringen
- Brukes når du *ikke trenger å endre originalen*

Eksempel:

```
for number in numbers:  
    print(number)
```

Teknikk 2:

```
for i in range(len(iterable)):
```

Syntaks:

```
for i in range(len(iterable)):  
    # gjør noe med iterable[i]
```

Forklaring:

- Itererer over indeksene
- Gir tilgang til både *posisjon* og *verdi* via iterable[i]
- Brukes når du skal *endre elementer* eller *trenger indeksen*

Eksempel:

```
for i in range(len(numbers)):  
    numbers[i] *= 2
```

numbers vil bli endret og ser slik ut etter at løkka har kjørt:
[6, 14, 4, 16, 8]

Teknikk 3:

```
for index, value in enumerate(iterable):
```

Syntaks:

```
for index, value in enumerate(iterable):
```

```
# gjør noe med index og value
```

Forklaring:

- Kombinerer indeks og verdi i én løkke
- Mer lesbart enn range(len(...))
- Kan også spesifisere startverdi: enumerate(iterable, **start=1**)

Eksempel:

```
for index, value in enumerate(numbers):
    numbers[i] *= value + 1
    print(f"Element {index} har verdien {value}")
```

Siden vi har indeks kan vi altså endre lista på samme måte som teknikk 2.

Kodeeksempelet under oppsummerer teknikkene som er beskrevet:

```
1: # file: sc_05_03.py
2: numbers = [3, 7, 2, 8, 4]
3:
4: for number in numbers:
5:     number = number * 2
6:     print(number, end=' ')
7: print()
8: print(numbers)
9:
10: for i in range(len(numbers)):
11:     numbers[i] = numbers[i] * 2
12: print(numbers)
13:
14: for index, value in enumerate(numbers):
15:     numbers[index] = value + 1
16: print(numbers)
```

Kjøring vil gi denne utskriften:

```
6 14 4 16 8
[3, 7, 2, 8, 4]
[6, 14, 4, 16, 8]
[7, 15, 5, 17, 9]
```

Linje 4 – 8 (ikke-destruktiv)

Itererer (gjennomløper) lista **numbers** fra start til slutt. Du må tenke deg at iterasjonsvariabelen **number** er en *kopi* av «nåværende» element i lista. Neste iterasjon gir neste element, etc.

Så, uansett hva vi gjør med elementet **number** så vil den ikke endre noe i lista **numbers**.

Utskriften, andre linje i utskriften over, bekrefter dette.

Linje 10 – 12

Her bruker vi **in range** konstruksjonen til å få ut en iterasjonsvariabel som kan brukes som en *indeks* i uttrykket både på høyre side og venstre side av

`numbers[i] = numbers[i] * 2`

Utskriften, linje tre, viser at vi har endret den originale lista.

Linje 14 – 16:

Her bruker vi **enumerate** konstruksjonen for å få ut en iterasjonsvariabel som egentlig består av *to ting*, en indekseringsvariabel **index**, samt selve elementet **value**:

`for index, value in enumerate(numbers):`

I kodelinja

`numbers[index] = value + 1`

«overskriver» vi gammel verdi på dette sted i lista med ny verdi ved hjelp av indeksen.

Vi ser av linje 4 i utskriften at vi har endret lista ved hjelp av denne konstruksjonen.

5.10.1 range konstruksjonen

Den innebygde funksjonen `range()` brukes til å generere en sekvens av heltall. Denne funksjonen brukes ofte i forbindelse med **for**-løkker, og gir en effektiv måte å iterere over tall på.

Syntaks:

`range(start, stop, step)`

Forklaring:

start: Det første tallet i sekvensen (inkludert)

stop: Sekvensen stopper *før* dette tallet (*ikke inkludert*)

step: Hvor mye sekvensen øker (*eller minker*) for hvert steg

Dersom **start** uteslates, settes den til 0. Dersom **step** uteslates, settes den til 1.

Eksempler:

```
for i in range(0, 5):
    print(i, end=" ") # utskrift: 0 1 2 3 4
```

```
for i in range(2, 10, 2):
    print(i, end=" ") # utskrift: 2 4 6 8
```

```
for i in range(5, 0, -1): # negativt steg
    print(i, end=" ") # utskrift: 5 4 3 2 1
```

Range uttrykk kan bli noe vanskelig å tolke, her et uttrykk med både negativ stopp-verdi og negativt steg:

```
for i in range(5, -1, -1):
    print(i, end=" ") # utskrift: 5 4 3 2 1 0
```

Dette siste uttrykkes tolkes slik:

start = 5: Vi begynner på tallet 5.

stop = -1: Iterasjonen stopper *før* den når -1.

step = -1: Vi går bakover, altså vi *minker* med 1 for hvert steg.

Viktig: `range()` inkluderer **start**, men ekskluderer **stop**. Det betyr at vi får med 0, men ikke -1.

Hvordan tolke **stop = -1?**

Når du bruker negativ **step**, må **stop** være **mindre enn start**, ellers får du en tom sekvens. **stop = -1** betyr at vi skal stoppe *før* vi når -1, altså siste verdi som inkluderes er 0.

Dette er en vanlig teknikk for å iterere baklengs over en liste eller indeksere fra slutten:

```
my_list = ['a', 'b', 'c', 'd', 'e', 'f']
for i in range(len(my_list) - 1, -1, -1):
    print(my_list[i], end=" ") # f e d c b a
```

range er et range-objekt

Rent teknisk er range i Python **ikke en liste**, men et **range-objekt** – en spesiell type iterator som genererer tall **på forespørsel** (lazy evaluation). Dette gjør range() svært minneeffektiv, spesielt ved store tallmengder.

Når vi skriver

```
r = range(0, 1000000)
```

...opprettes det **ikke** en liste med én million tall. I stedet opprettes et objekt av typen range, som **vet hvordan** det skal generere tallene når du ber om dem.

Vi kan lage en liste med tall fra en range slik:

```
list1 = list(range(5)) # [0, 1, 2, 3, 4]
```

...men merk, dette lager **ikke** en liste med tall, men en liste med *ett range objekt*:

```
list1 = [range(5)] # [range(0, 5)]
```

Fordeler med range:

- **Minneeffektiv:** Bruker konstant minne, uansett hvor stor sekvensen er
- **Rask:** Genererer tallene kun når de trengs
- **Immutable:** Kan ikke endres etter at det er opprettet

5.10.2 enumerate()

enumerate() lar deg iterere over en liste samtidig som du får både indeks og verdi.

Du kan også spesifisere hvilken verdi indeksen skal starte på; – for eksempel 1 i stedet for standardverdien 0.

Her er et eksempel som skriver ut månedene med nummerering fra 1:

```
months = ["January", "February", "March", "April", "May", "June",
          "July", "August", "September", "October", "November", "December"]
for number, month in enumerate(months, start=1):
    print(f"{number}: {month}")
```

Utskrift:

```
1: January
2: February
3: March
...
12: December
```

På samme måte som `range` returnerer `enumerate` et spesielt objekt, et `enumerator` objekt. Dette har de samme fordelene som er nevnt for `range`.

5.10.3 Kopiering av liste med for – løkke

Vi kan bruke `for` - løkka til å lage en kopi av en liste, men obs dette er også en *shallow copy*:

```
original_list = [1, 2, 3]
copy_list = []

for element in original_list:
    copy_list.append(element)

print("Original:", original_list)
print("Copy:", copy_list)
for i in range(len(list1)):
    print(id(list1[i]))
    print(id(list5[i]))
```

Utskrift:

```
Original: [1, 2, 3]
Copy: [1, 2, 3,]
1616178053360
```

```
1616178053360
1616178053392
1616178053392
1616178053424
1616178053424
```

Vi ser at listene refererer samme objekter. Bruk `copy.deepcopy()` for å få en fullstendig kopiering, hvis det er nødvendig.

5.10.4 Scope av variabler i løkker

Scope begrepet handler om når en variabel er synlig / tilgjengelig for kode. Regler for synlighet i løkker og funksjoner og annet er beskrevet i kapittel <[må fylles i](#)>.

5.11 Slicing

5.11.1 Generelt

Slicing er en kraftig teknikk i Python som lar deg hente ut deler av en sekvens – som for eksempel en streng, liste eller tuple – ved å spesifisere et startpunkt, et sluttpunkt og eventuelt et steg. Dette gjør det enkelt å ta ut deler av data uten behov for løkker eller ekstra logikk.

Den generelle syntaksen for slicing er:

sekvens[start:slutt:steg]

- **start**: indeksen der slicing begynner (inkludert)
- **slutt**: indeksen der slicing slutter (ikke inkludert)
- **steg**: hvor mange elementer man hopper over (kan være negativ for å gå baklengs)

Hvis noen av disse verdiene utelates, eller at `None` brukes, så anvendes *standardverdier*:

- **start** → 0 (eller siste element hvis **steg** < 0)
- **slutt** → lengden på sekvensen (eller før første element hvis **steg** < 0)
- **steg** → 1

Merk:

Dersom du bruker ulovlige verdier i slicing vil ikke programmet stoppe på grunn av index out of bound, slik som du får ved vanlig indeksering . Slicing er tilgivende og erstatter ulovlige verdier med lovlige. Pass derfor på at du får det ønskede resultatet i forhold til det du forventer...

Her er hva som skjer ved ulovlige verdier ved vanlig indeksering (første eksempel) og slicing:

s = "Helloworld"

Tilgangstype Feil ved ugyldig indeks?

- | | |
|-----------|----------------------------------|
| s[100] | Ja, IndexError |
| s[0:100] | Nei, returnerer det som er mulig |
| s[-100:3] | Nei, starter fra begynnelsen |
| s[10:20] | Nei, returnerer tom streng |

5.11.2 Slicing kan utføre på sequences

Slicing i Python kan bli utført på datatyper som tilhører gruppen sequences.

Dette er:

1. Strenger (str)
2. Lister (list)
3. Tuples (tuple)
4. Bytes (bytes)
5. Bytearrays (bytearray)
6. Ranges (range)

En sequence er som vi har sett tidligere en ordnet samling av elementer.

**Reglene for slicing er akkurat de samme for alle disse typene.
I eksemplene nedenfor bruker vi en streng (str) for å demonstrere hvordan slicing fungerer.**

```
1: # file: sc_05_05_slicing_demo.py
2: s = "Helloworld"
3:
4: print(s[0:5])          # "Hello" - fra indeks 0 til 4
```

```

5: print(s[:])           # "Helloworld" - hele strengen
6: print(s[::-1])        # "Helloworld" - hele strengen
7: print(s[None:-1])     # "Helloworld" - None tolkes som
8:                      # standardverdi, dvs 0
9: print(s[0:len(s)])    # "Helloworld" - hele strengen
10: print(s[:None])       # "Helloworld" - None tolkes som
11:                      # standardverdi, dvs len(s)
12:
13: print(s[0:5:2])       # "Hlo" - annethvert tegn fra
14:                      # indeks 0 til 4
15: print(s[::-1])        # "dlrowolleH" - reverserer hele
16:                      # strengen, standardverdier brukes
17: print(s[None:None:-1]) # "dlrowolleH" - reverserer hele
18:                      # strengen, standardverdier brukes
19: print(s[5:0:-1])      # "wolle" - baklengs fra indeks 5
20:                      # til 1
21: print(s[5::-1])       # "olleH" - baklengs fra indeks 5
22:                      # til starten
23:
24: # slicing på en liste, samme regler som for strenger
25: list1 = ['H', 'e', 'l', 'l', 'o', 'w', 'o', 'r', 'l', 'd']
26: print(list1[::-1]) # [d', 'l', 'r', 'o', 'w', 'o', 'l', 'l', 'e', 'H']

```

5.11.3 Reversering ved hjelp av slicing

I linje 15 og 17 reverseres en streng:

```

print(s[::-1]) # "dlrowolleH"
print(s[None:None:-1]) # "dlrowolleH"

```

Dette er et såkalt *Python idiom*: Det er kort, effektivt og uformelt ansett som "den Python-aktige måten" å reversere en sequence.

Kommer du til et jobbintervju bør du helst kunne dette «trikset» 😊
 Idiomet `s[::-1]` for å reversere en streng er elegant og effektivt, men det kan være litt vanskelig å forstå, – og det har alt å gjøre med **standardverdiene i slicing**.

Hvorfor kan `s[::-1]` være forvirrende?

To slicing-parametere er *implisitte*:

`s[start:stop:-1]` → her er både `start` og `stop` utelatt, og bare `step = -1` er spesifisert.

Det betyr at man må vite hva Python gjør når start og stop er utelatt, hvilket er det samme som å oppgi None.

Negativt steg snur retningen:

Når `step = -1`, tolker Python slicing som å "gå baklengs".
Dette snur også hvordan start og stop tolkes.

Standardverdiene endres med retningen:

Hvis `step` er *positivt*:

```
start = 0 (start fra begynnelsen)
stop = len(s) (slutt ved enden)
```

Hvis `step` er *negativt*:

```
start → siste element
stop → før første element
step → 1
```

MERK at for reversering av strengen kan du ikke oppgi noe annet enn `None` eller ingenting som `stop` index...

En god regel er at hvis du skal reversere ved hjelp av slicing, bruk slicing slik:

`[::-1]`

uten å fylle ut verdier for `start` og `stop`.

Som nevnt, slicing syntaksen fungerer på alle sekvenser, så dersom vi har en *liste*:

```
list1 = ['H', 'e', 'l', 'l', 'o', 'w', 'o', 'r', 'l', 'd']
```

så vil reversering av lista ha samme effekt som reversering av en streng:

```
print(list1[::-1]) # ['d', 'l', 'r', 'o', 'w', 'o', 'l', 'l', 'e', 'H']
```

Python håndterer også slicing med *indekser utenfor grensene*:

```
s = "Hello world"
s1 = s[-100:100]
print(s1) # "Hello world"
```

Det er verd å merke seg at slicing faktisk bruker et eget objekt i bakgrunnen:

```
s = "Hello world"
slice_obj = slice(None, None, -1)
```

```
print(slice_obj.start, slice_obj.stop, slice_obj.step) # None,
# None,-1
print(s[slice_obj]) # "dlrow olleH"
```

Siden vi ikke har lært mye om objekter; - ta det over som en informasjon, og kom tilbake når du har lært om klasser og objekter.
Poenget er at et uttrykk som [None :None : -1] vil bli oversatt av Python oversetteren til den objektorienterte utgaven over.

5.11.4 Et slicing uttrykk kan oppstre på venstre side av en tilordning

```
1: # file: sc_05_08.py
2: # et slicing uttrykk kan stå på venstre side av en tilordning
3:
4: list1 = [0, 1, 2, 3, 4, 5]
5: list1[1:4] = [10, 11, 12] # erstatter elementer
6: print(list1) # [0, 10, 11, 12, 5]
7:
8: list1[2:4] = [] # fjerner elementer
9: print(list1) # [0, 10, 5]
10:
11: list1[1:1] = [99, 100] # Setter inn to elementer før indeks 1
12: print(list1) # [0, 99, 100, 10, 5]
```

Se kommentarer i koden for linjene 4 – 9.

Linje 11 krever en nøyere forklaring:

Når du bruker slicing med *samme start- og sluttindeks*, som i list1[1:1], så peker det på *et tomt område foran den indeksen*. Det betyr at du setter inn elementer *før* det som allerede ligger på den indeksen. Dette er ikke nødvendigvis intuitivt første gang man ser det, men det følger logikken i hvordan slicing fungerer i Python:

list1[a:b] refererer til elementene fra indeks a opp til, men ikke med b.

Hvis a == b, er det ingen elementer i området – men det er fortsatt en gyldig plassering for innsetting.

En annen måte å se det på:

Tenk deg at elementene i listen står på en rekke, og mellom hvert element er det en "sprekk" der du kan sette inn noe nytt. list1[1:1] er akkurat den sprekken mellom element 0 og 1.

Denne regelen er kun relevant når slicing uttrykket står på venstre side av tilordningen 😊

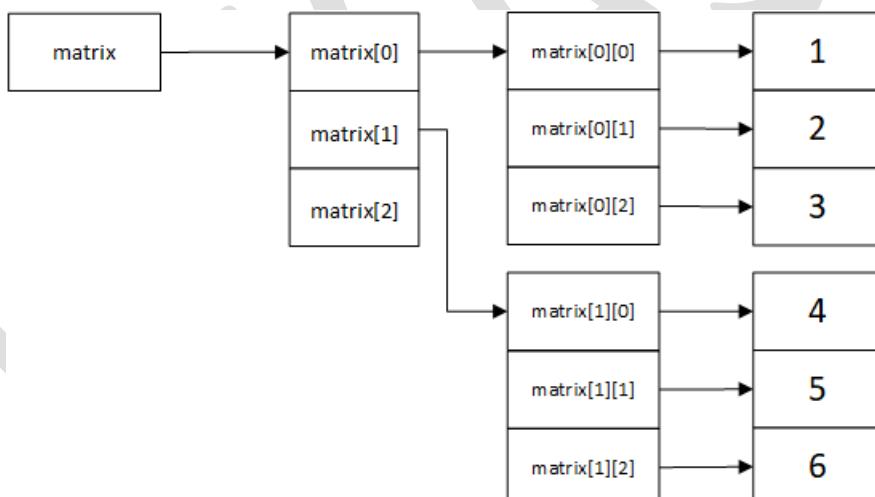
5.12 Flerdimensjonale lister

Lister i Python kan inneholde andre lister som elementer. Dette gjør det mulig å lage *flerdimensjonale datastrukturer*, som for eksempel matriser (2D-lister), tabeller og mer komplekse datastrukturer.

En todimensjonal liste er en liste hvor hvert element er en annen liste. Dette kan for eksempel brukes til å representer en matrise med heltall:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

`matrix` er en todimensjonal liste med referanser til nye endimensjonale lister, som har referanser til tallene, se Figur 5.6



Figur 5.6 En todimensjonal liste

Vi får tilgang til et element ved å bruke to indekser:

```
value = matrix[1][2] # Gir 6
```

... eller hele den første listen ved å bruke bare en indeks:

```
first_row = matrix[0]
```

Det er vanlig å kalle første dimensjon i en todimensjonal matrise for *rad*, og den andre for *kolonne*.

5.12.1 Iterere over flerdimensjonale lister, og ragged lists

Når vi itererer gjennom flerdimensjonale lister trenger vi løkker i løkker. Løkkekonstruksjonen under vil fungere også for *ragged arrays*, at sublistene har forskjellig lengde:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

for row in matrix:
    for element in row:
        print(element, end=' ')
    print() # linjeskift etter hver rad
```

Utskrift:

```
1 2 3
4 5 6
7 8 9
```

`enumerate` konstruksjonen kan være nyttig å bruke når en ønsker å skrive ut indeksene til lista:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

for row_index, row in enumerate(matrix):
    for col_index, value in enumerate(row):
        print(f"matrix[{row_index}][{col_index}] = {value}")
```

Utskrift:

```
matrix[0][0] = 1
matrix[0][1] = 2
matrix[0][2] = 3
...
matrix[2][2] = 9
```

Lister i Python kan være «ragged», det vi si dimensjonene i sublistene behøver ikke være like.

`enumerate` konstruksjonen er hendig også her; - `enumerate` tilpasser seg, som den ordinære for løkka, til lengden av den aktuelle (sub-) lista.

Her vist med et eksempel hvor vi også **endrer** lista.:

```
matrix = [
    [1, 2],
    [3, 4, 5],
    [6]
]

for i, row in enumerate(matrix):
    for j, value in enumerate(row):
        matrix[i][j] = value * 10

print(matrix)
```

Utskrift:

`[[10, 20], [30, 40, 50], [60]]`

5.12.2 Generisk kode for iterering av ukjent antall dimensjoner

Generelt er det slik at det må være like mange nøstede for-løkker som det er nivåer i den flerdimensjonale tabellen, for å få iterert over hele strukturen.

Det **er** mulig å lage *generisk, dynamisk kode* som finner ut hvor mange dimensjoner det er i en flerdimensjonal liste; - vi kan bruke *rekursjon* for å finne ut hvor «dyp» en listestruktur er. For kuriositetens skyld, her er koden. Hvis alt for uforståelig, kom tilbake når vi har lært om funksjoner og rekursjon (en metode kaller seg selv) 😊 :

```
def traverse(data, level=0):
    for item in data:
        if isinstance(item, list): # er det en liste?
            traverse(item, level + 1) # rekursivt kall
        else:
            print(" " * level + str(item))
```

```
# Eksempel på bruk:
nested_list = [1, [2, 3], [[4, 5], 6], [[[7]]]]
traverse(nested_list)
```

Utskrift:

```
1
2
3
4
5
6
7
```

`isinstance()` er en systemfunksjon i Python som returnerer True hvis første argument er av typen til andre argument (her: `list` datatypen).

5.13 List comprehensions

5.13.1 Comprehensions er en kortform av for løkke

Comprehensions brukes til å skrive kompakt kode, og er en kortform av for-løkker (iblandet if-setninger). Comprehensions kan lages både for lists, tuples, sets og dictionaries.

Comprehensions kan bli komplisert og vanskelig å lese, så bruk det med varsomhet. Det anbefales ikke å bruke mer enn to for-løkker i en enkelt comprehension.

Comprehensions er i prinsippet «syntactic sugar»; - en språk-konstruksjon som supplerer «det vanlige», i dette tilfellet for-løkker.

For å se sammenhengen mellom «normal» kode med for- løkker og tilsvarende comprehensions, her er noen eksempler:

Lage en liste hvor hvert element «oppheyes i andre» («squared»), først med for-loop deretter med comprehension:

```
# list with squares - for-loop & append
squares = []
for x in range (10):
    squares.append(x*x)
print(squares) # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# list with squares - comprehension
squares = [x * x for x in range(10)]
print(squares) # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Vi ser at *tre linjer med kode* kan erstattes av *en kodelinje*.

Legg også merke til at `append` statementet er implisitt; - det at uttrykket står inni `[]` parenteser betyr at det skal lages et nytt item (`append`) til lista.

Den generelle syntaksen for en list comprehension i sin enkleste form er:

`values = [expression for item in collection]`

... som tilsvarer, skrevet med `for`-løkke:

```
values = []
for item in collection:
    values.append(expression)
```

Vi ser at `append` er *implisitt* i comprehension konstruksjonen, og at `expression` er plassert *før* for-løkka inni comprehension'en, ikke etter, som i den ordinære `for`-løkka.

Vi kan også inkludere et «filter» i comprehension uttrykket:

```
odd_squares = [x * x for x in range(10) if x % 2 != 0]
print(odd_squares) # [1, 9, 25, 49, 81]
```

...som tilsvarer følgende kode med `for`-løkke:

```
odd_squares = []
for x in range(10):
    if x % 2 != 0:
        odd_squares.append(x * x)
```

```
print(odd_squares) # [1, 9, 25, 49, 81]
```

Den generelle syntaksen for en list comprehension med «filter» (en `if` setning) blir:

```
values = [expression for item in collection if condition]
```

...eller skrevet slik at en bedre ser de enkelte delene av uttrykket:

```
values = [expression
          for item in collection
          if condition]
```

...og skrevet som en for-løkke:

```
values = []
for item in collection:
    if condition:
        values.append(expression)
```

5.13.2 Løkke i løkke skrevet som comprehension

Se på denne løkke- i løkke konstruksjonen:

```
result = []
for x in range(3):
    for y in range(2):
        result.append(x * y)
print(result) # [0, 0, 0, 1, 0, 2]
```

Skrevet som en comprehension vil dette bli:

```
result = [x * y for x in range(3) for y in range(2)]
print(result) # [0, 0, 0, 1, 0, 2]
```

Regel:

Den første / ytterste for løkka (`for x in range(3)`) kommer først i comprehension uttrykket, og den innerste løkka (`for y in range(2)`) kommer sist.

I en list comprehension som den over er det den *første* for-løkken (`for x in range(3)`) som starter først.

Men det er den *siste* for-løkken (`for y in range(2)`) som "går raskest" (innerst), akkurat som i nærmeste for-løkker.

Merk at i comprehensions skrevet for flerdimensjonale lister, så vil parantesene som bestemmer dimensjonen overstyre eksekveringsrekkefølgen av løkkene. Se egne eksempler senere.

Generelt tips når du skal skrive en comprehension:

Det kan være lurt å skrive en vanlig for-løkke først, og deretter oversette den til en comprehension for bedre lesbarhet.

5.13.3 List comprehensions: flerdimensjonale lister

Kodeeksempelet viser hvordan vi lager en *tedimensjonal* liste og fyller den med verdier. Det vises tre måter å gjøre dette på:

1. Ved initialisering med literaler («Hardkoding»)
2. For løkker med `append()`
3. List comprehension

```
# 1. Direkte initialisering
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print(matrix) # [[1, 2, 3], [4, 5, 6], [7, 8, 9]]


# 2. To for-løkker og append()
# Ytterste løkke lager hver rad, innerste løkke fyller
# hver rad med verdier.
NUM_ROWS = 3
NUM_COLS = 3
matrix = []
```

```

for i in range(NUM_ROWS):
    row = []
    for j in range(NUM_COLS):
        row.append(i * NUM_COLS + j + 1) # Fyller med
    tall fra 1 til 9
    matrix.append(row)
print(matrix) # [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# 3. Bruke list comprehension
matrix = [[i * NUM_COLS + j + 1 for j in range(NUM_COLS)]
for i in range(NUM_ROWS)]
print(matrix) # [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

```

Parentesene og plasseringen av for-løkkene i list comprehension kan virke forvirrende.

I en nøstet list comprehension som:

```
matrix = [[i * NUM_COLS + j + 1 for j in range(NUM_COLS)]
for i in range(NUM_ROWS)]
```

ser det ut som den ytterste løkken (`for i in range(NUM_ROWS)`) står / evalueres sist, fordi den står lengst til høyre.

Men den evalueres *først*, akkurat som i tradisjonelle nøstede for-løkker:

Parentesene rundt den innerste comprehension lager en ny liste for hver `i`, og det er derfor rekkefølgen blir riktig.

Så selv om syntaksen ser litt "bakvendt" ut, er rekkefølgen på utførelsen den samme som i vanlige for-løkker.

Huskeregel:

Plasser den indre for-løkka i den indre dimensjonen, altså:

```
matrix = [[ indre løkke ] ytre løkke]
```

5.13.4 List comprehension: Enda et eksempel

Følgende for-løkke konstruksjon fyller en todimensjonal, $N \times N$ liste med `False` verdier:

```
grid = []
for k in range(N):
```