Kapittel 5: `list`, og `for` løkka

Del 1, klassen `list`

`list` og `for`-loop hører sammen!

- 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.

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?
```

Håpløst å behandle 10 enkeltstående variabler under ett..

- 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)
```

Utskrift:
Summen av de innleste tallene er: 365
Tallene, sortert: [0, 2, 4, 5, 6, 45, 67, 71, 78, 87]

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

```
print("Tallene, sortert:", heltall_liste)
```

Utskrift:

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

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 programmererne 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].

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

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.*

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.

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

Opprette en liste og legge til elementer

```
 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) # [1]
19: number_list.append(2) # [1,2]
20: number_list2.append(10) # [10]
```

En liste kan endres med indeksering

```
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']
```

Tilsvarende var ikke mulig med `str`..

Vær obs på denne: Her brukes list constructoren til å ta imot en *streng*. List constructoren aksepterer **ett** itererbart objekt, og tekst er et slikt, nemlig et str objekt. Når constructoren itererer over tekst, 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]

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:

```
navn = ["Anna", "John", "Clara"]
```

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"]`

Negative indekser kan brukes:

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

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

Operasjoner på lister

En liste er en *sequence*, som vi kan utføre generelle sequence operasjoner på

I tillegg har `list` klassen egne medlemsmetoder

Sequence operasjoner på lister

Gitt disse listene:

seq1 = [1, 2, 3]
seq2 = [4, 5, 6]

Så kan vi utføre

Operasjon	Beskrivelse	Eksempel
`len(seq1)`	Antall elementer i sekvensen	`len(seq1) → 3`
`seq1[i]`	Hente element på indeks `i`	`seq1[1] → 2`
`seq1[i:j]`	Slicing – delsekvens fra `i` til `j-1`	`seq1[0:2] → [1, 2]`
`seq1 + seq2`	Konkatinering (sammenslåing)	`seq1 + seq2 → [1, 2, 3, 4, 5, 6]`
`seq1 * 2`	Multiplikasjon	`seq1 * 2 → [1, 2, 3, 1, 2, 3]`
`x in seq1`	Sjekk om `x` finnes i sekvensen	`if 2 in seq1 → True`
`min(seq2) / max(seq2)`	Minimum / maksimum verdi	`min(seq2) → 4` `max(seq2) → 6`
`sum(seq1)`	Summerer tall i sekvensen	`sum(seq1) → 6`

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

Sequence operasjoner på lister (forts)

Gitt disse listene:

seq1 = [1, 2, 3]
seq2 = [4, 5, 6]

Så kan vi utføre

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

Operasjon	Beskrivelse	Eksempel
`sorted(seq2)`	Returnerer sortert ***kopi*** (i motsetning til `list` medlemsmetoden `sort()`, som sorterer in-place og returnerer `None`)	`sorted(seq2) → [4, 5, 6]`
`reversed(seq1)`	Returnerer reversert iterator	`list(reversed(seq1)) → [3, 2, 1]`
`range(start, stop[, step])`	Genererer en tallsekvens	`range(1, 5) → [1, 2, 3, 4]`
`enumerate(seq1)`	Returnerer (indeks, verdi)-par	`list(enumerate(seq1)) → [(0, 1), (1, 2), (2, 3)]`
`zip(seq1, seq2)`	Slår sammen elementer fra to sekvenser	`list(zip(seq1, seq2)) → [(1, 4), (2, 5), (3, 6)]`
`seq1 == [1, 2, 3]`	Sammenligning av sekvenser	`True`
`seq1 < seq2`	Leksikografisk sammenligning	`True (fordi 1 < 4)`
`sorted(seq2)`	Returnerer sortert ***kopi*** (i motsetning til `list` medlemsmetoden `sort()`, som sorterer in-place og returnerer `None`)	`sorted(seq2) → [4, 5, 6]`

Operasjoner på lister: medlemsfunksjoner

- Forrige lysbilder 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']
```

Først, la oss gjøre `dir()` på list klassen…

```
>>> dir(list)
['__add__', '__class__', '__class_getitem__', '__contains__',
'__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>>
```

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)  # []
```

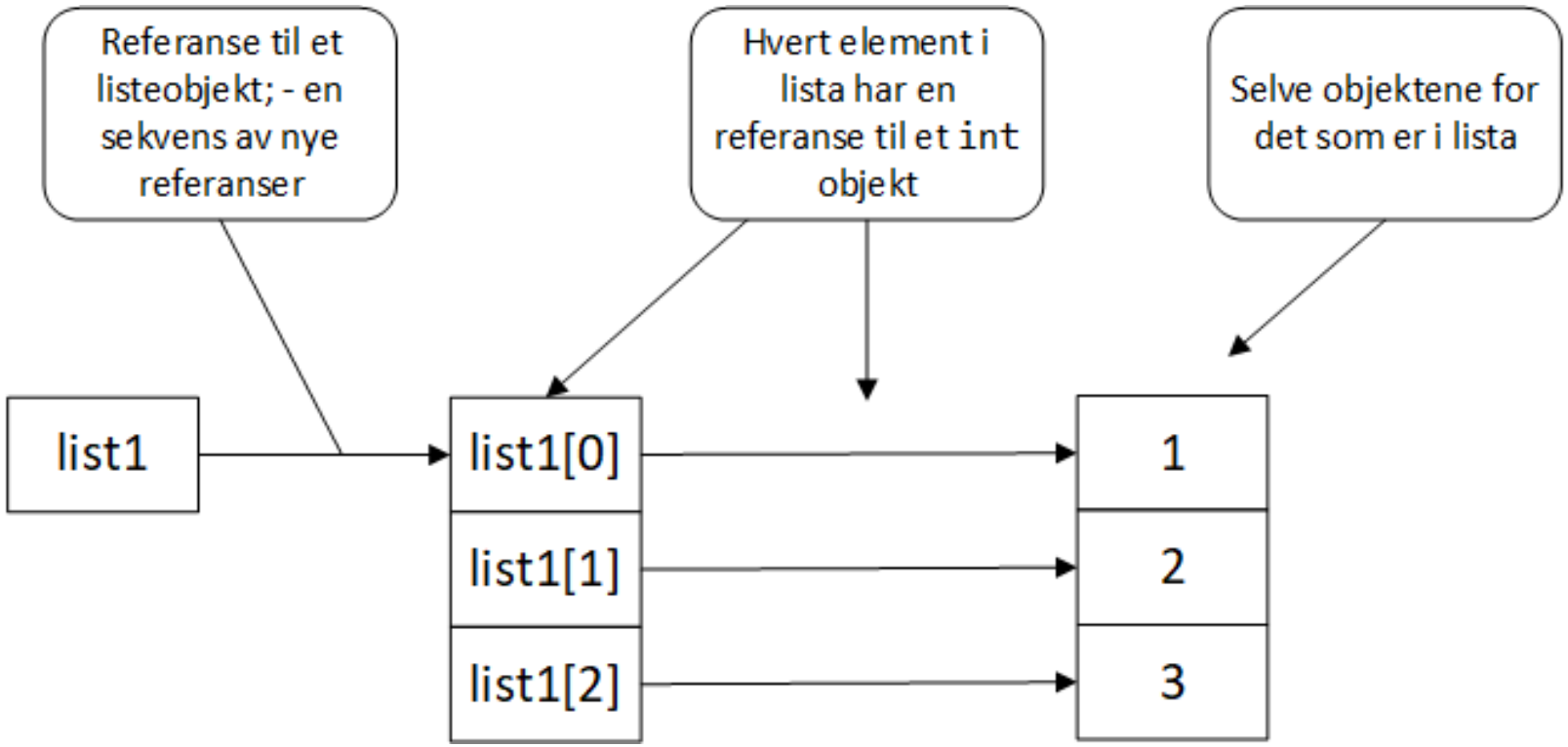
En liste er en liste av referanser

- 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.
- Se neste lysbilde..

```
list1 = [1, 2, 3]
```


Referanse til et listeobjekt; - en sekvens av nye referanser
Hvert element i lista har en referanse til et int objekt
Selve objektene for det som er i lista
list1
list1[0]
list1[1]
list1[2]
1
2
3

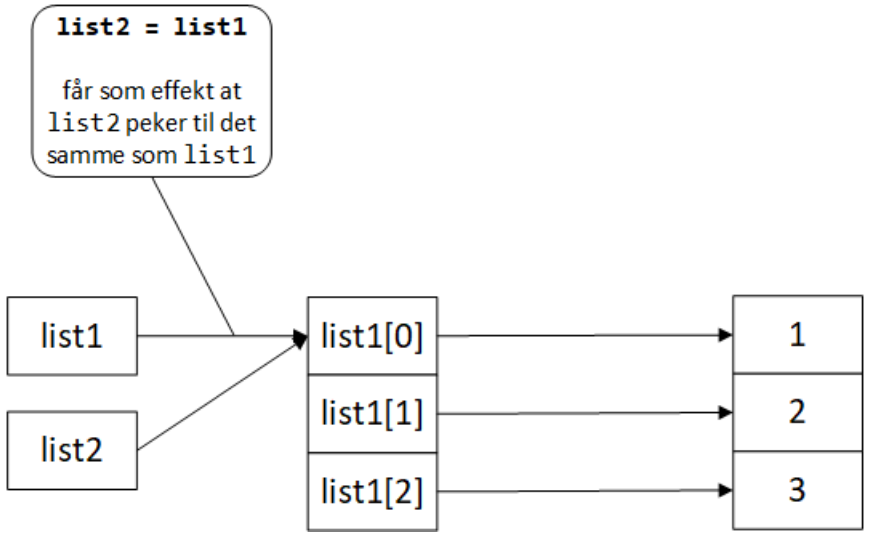
Kopiere lister: dette er IKKE kopiering

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 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
>>>
```

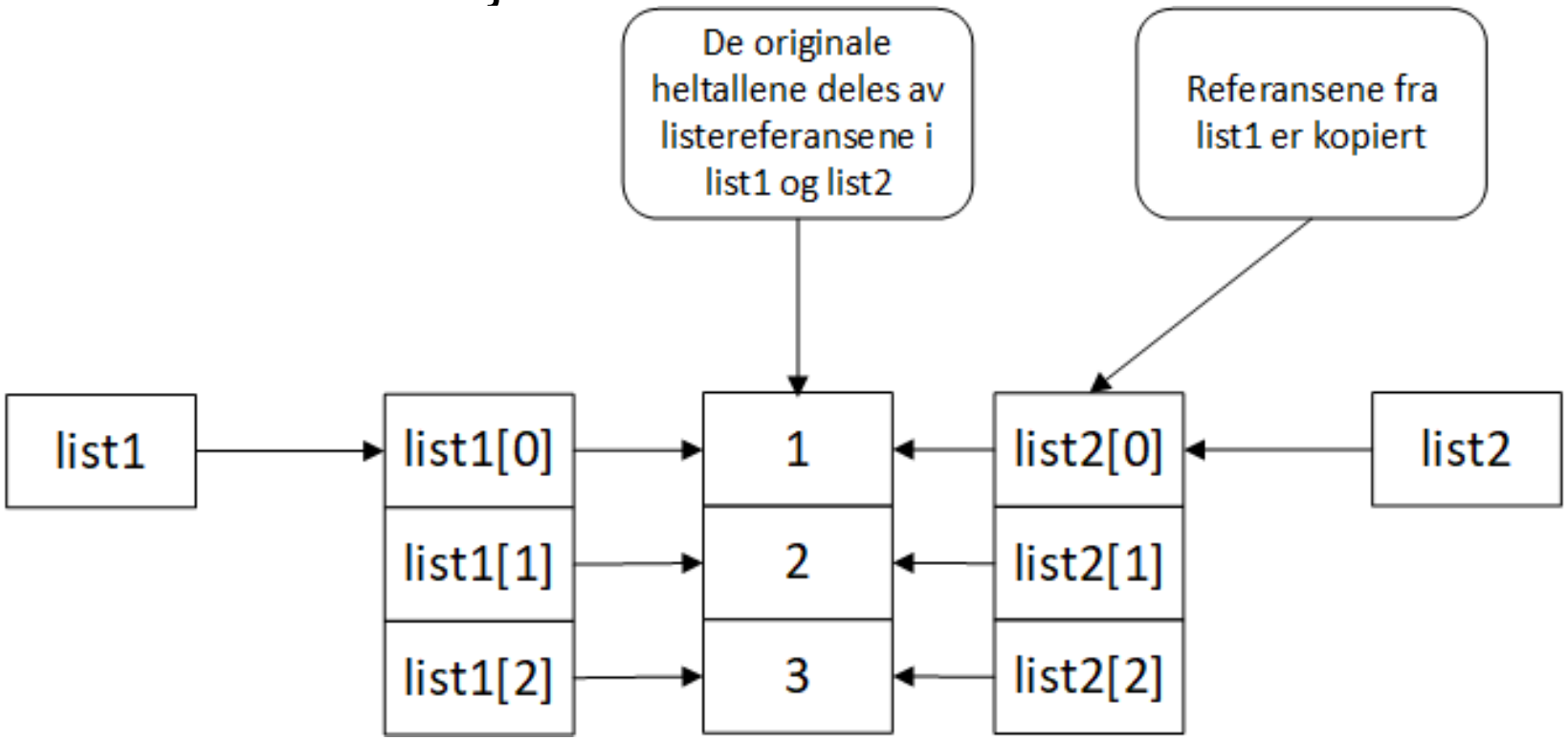


Kopiere lister: shallow copy

- 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 nye 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.

Kopiere lister: shallow copy

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



Shallow copy med slicing

```
01: # file : sc_05_04.py
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]
```

Shallow copy med comprehension

```
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]
```

Shallow copy med +

```
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]
```

Shallow copy med list klassen sin copy()

```
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]
```

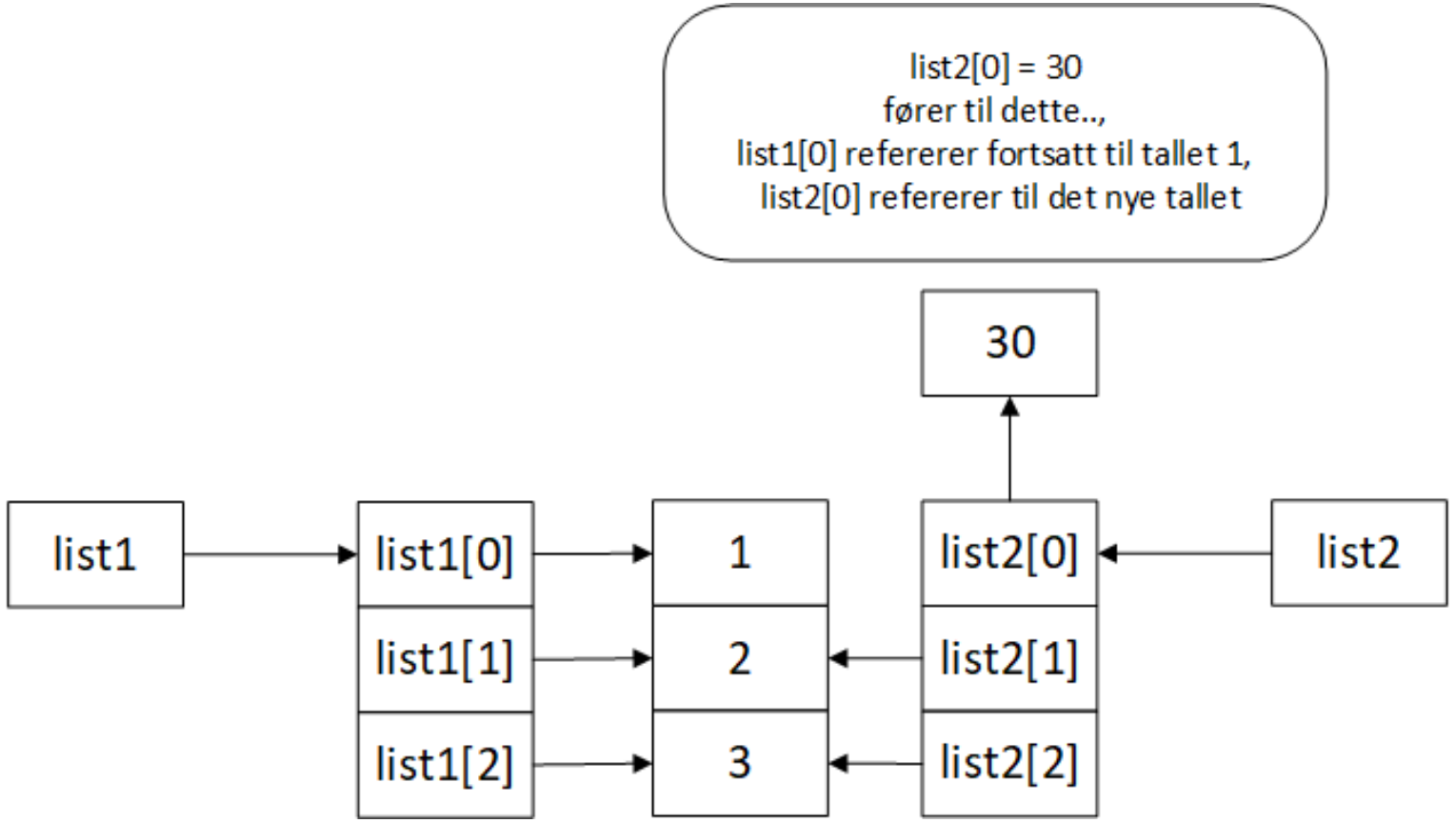
Shallow copy med copy.copy()

```
02: import copy
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]
```

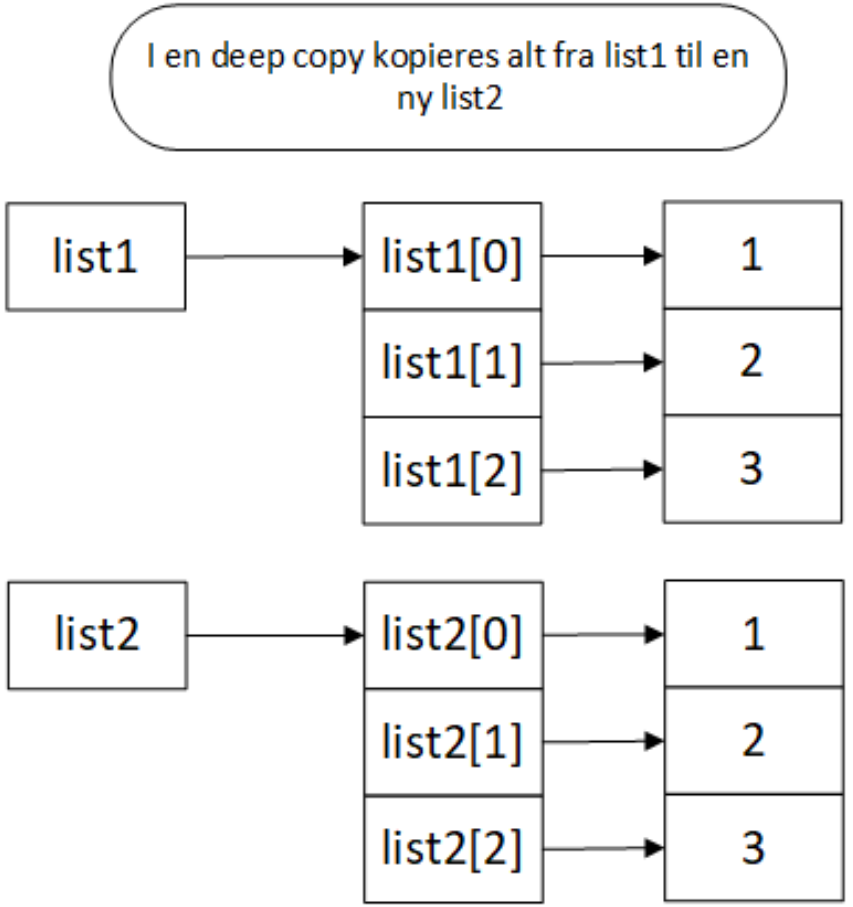
Deep copy...

```
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], men se figur!
```

Situasjon etter shallow copy



Situasjon etter 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.

Anbefalt måte å gjøre en shallow copy

- Anbefalt metode av de vi har sett på er `list` klassen sin `copy()`

```
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)
```