Objekter og referanser – noen korte eksempler

La oss se på noen eksempler på hva som kan skje når vi jobber med lister.

Lister er objekter, og de er i tillegg *muterbare* (mutable). Det vil si at vi kan endre på selve objektet. Dette gjelder lister, ordbøker, og alle klassene vi lager selv, men ikke blant annet strenger og tall.

Det er derfor viktig å huske at to variabelnavn kan peke på samme objekt (samme adresse), og hvis man endrer objektet gjennom den ene variabelen, så vil den andre variabelen også gjenspeile de samme endringene, siden den peker på det samme objektet.

Vi skal gjøre noen tester med "==" som tester innholdslikhet, og kodeordet *is* som tester om to variabler peker på samme objekt. Akkurat hvilke egenskaper "==" refererer til for klasser vi lager selv, kan vi bestemme selv ved å endre den magiske metoden __eq__, som dere så i ukeoppgavene i en tidligere seminartime.

Eksempel

Kjøringen er fra et interaktivt Python-miljø, så vi ser verdiene til variablene ved å skrive dem, uten print().

Vi oppretter tre nye lister. a og c har likt innhold, mens b har annerledes innhold.

```
>>> a = [1,2,3]
>>> b = [2,3,4]
>>> c = [1,2,3]
```

Så tester vi om innholdet i a og b er likt. Det blir som forventet *False*.

```
>>> a == b
False
```

Hvis vi sammenlikner a og c får vi *True*. Det er fordi listene er like lange og har like elementer.

```
>>> a == c
True
```

Hvis vi undersøker om a og b peker på samme objekt, eller om a og c peker på samme objekt, får vi feil i begge tilfellene. Vi har opprettet objektene separat, ingen er felles.

```
>>> a is b
False
>>> a is c
False
```

Men hvis vi så oppretter en ny variabel, og sier d = a, så sier vi at d skal *peke på* det samme som a peker på.

```
>>> d = a
>>> d
[1, 2, 3]
```

d og a har derfor ikke bare samme innhold, men de peker altså på helt samme objekt. Det kan vi teste.

```
>>> d == a
True
>>> d is a
True
```

Det som er viktig å merke seg da, er at hvis vi gjør endringer med d, så vil disse endringene også gjelde a. Her ser vi at hvis bruker .append() på d, og ser hva verdien til a er, så ser vi at endringene har skjedd i både a og d. Husk at . er dott-notasjon, og viser at vi bruker en metode på objektet d. Mange av dere har brukt *del*. *del* har en annen syntaks, vi bruker ikke dott-notasjon her, men det endrer også selve objektet.

```
>>> d.append(7)
>>> a
[1, 2, 3, 7]
>>> d
[1, 2, 3, 7]
>>> del a[1]
>>> d
[1, 3, 7]
>>> a
[1, 3, 7]
```

En siste ting (strengt talt ikke pensum), er at hvis vi tar et stykke (slice) av ei liste, så blir det laget et nytt liste-objekt med de nye verdiene. Her tar vi de to første verdiene til a (som er [1,3,7]) og oppretter et nytt liste-objekt med disse verdiene, og lagrer adressen i e.

```
>>> e = a[:2]
>>> e
[1, 3]
>>> e is a
False
```