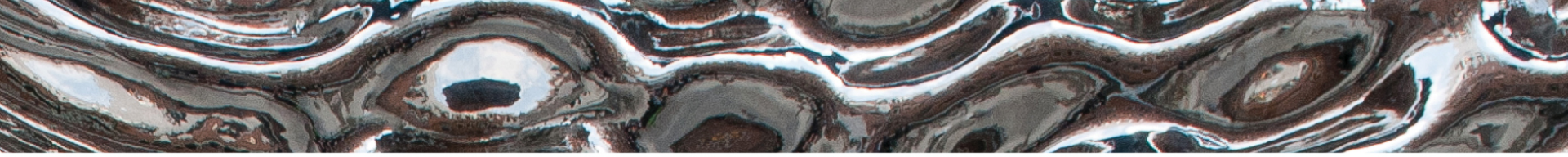


UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Magiske metoder i Python

Magiske metoder

- Vi kjører en metode ved å skrive navnet på den: per.`spisBanan()`
- Magiske metoder i Python er metoder som **kjøres uten at vi skriver navnet på metoden**!
  - For eksempel `__init__(self, navn)` som kjører i det vi oppretter et nytt objekt av klassen

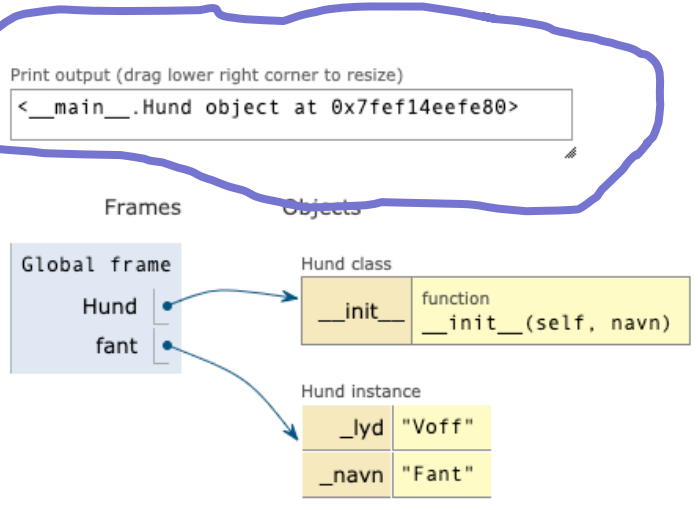
Har du prøvd å printe ut et objekt av en egendefinert klasse?

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  class Hund:
2      def __init__(self, navn):
3          self._navn = navn
4          self._lyd = "Voff"
5
6  fant = Hund("Fant")
7
8  print(fant)
9
10
11
12
```

Print output (drag lower right corner to resize)

```
<__main__.Hund object at 0x7fef14eefe80>
```



Den magiske metoden `__str__(self)`

- Hvis vi definerer metoden `__str__(self)` i klassen brukes denne på magisk vis når vi skriver print(objekt)

- Magiske metoder i Python er metoder som **kjøres uten at vi skriver navnet på metoden**!

- I metoden definerer vi hvordan utskrift av objektene vil bli

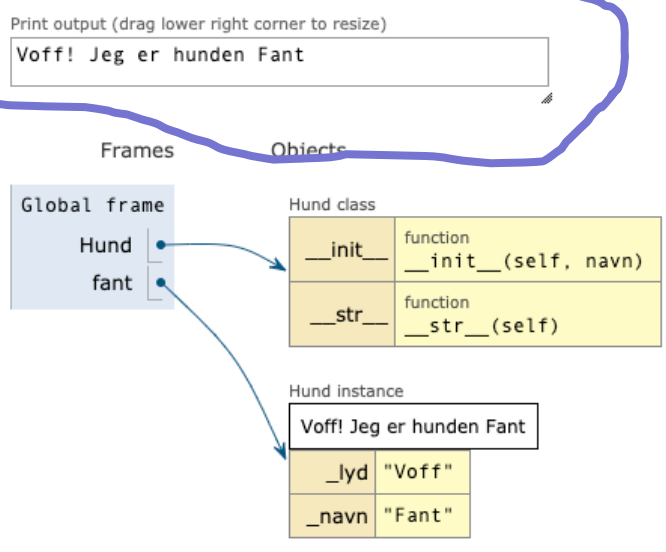
Den magiske metoden `__str__(self)`

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  class Hund:
2      def __init__(self, navn):
3          self._navn = navn
4          self._lyd = "Voff"
5
6      def __str__(self):
7          return f"{self._lyd}! Jeg er hunden {self._navn}"
8
9  fant = Hund("Fant")
10
11 print(fant)
12
13
14
15
16
```

Print output (drag lower right corner to resize)

```
Voff! Jeg er hunden Fant
```



Sjekke om noe er likt noe annet

- For tall og tekst er dette ganske enkelt

```
print ("Børre" == "Børre")

navn1 = "Hal"
navn2 = "Hal"

print(navn1 == navn2)

t1 = 200.5
t2 = 200.5

print(t1 == t2)
```

Print output (drag lower right corner to resize)

```
True
True
True
```

Sjekke om noe er likt noe annet

- For en liste fungerer det også fint

```
print( ["A", "B", "C"] == ["A", "B", "C"] )
print( ["A", "B", "C"] == ["A", "B", "C", "D"] )


mineSvar = [23, 43, 21]
fasit = [23, 43, 21]

print(mineSvar == fasit)

fasit.append(32)
print(mineSvar == fasit)
```

Print output (drag lower right corner to resize)

```
True
False
True
False
```

Sjekke om noe er likt noe annet

- Også for en ordbok (dictionary)

```
poststeder1 = {
    "1350" : "Lommedalen",
    "1405" : "Langhus",
    "7800" : "Namsos"
}

poststeder2 = {
    "1350" : "Lommedalen",
    "1405" : "Langhus",
    "7800" : "Namsos"
}

print( poststeder1 == poststeder2)
```

Print output (drag lower right corner to resize)

```
True
```

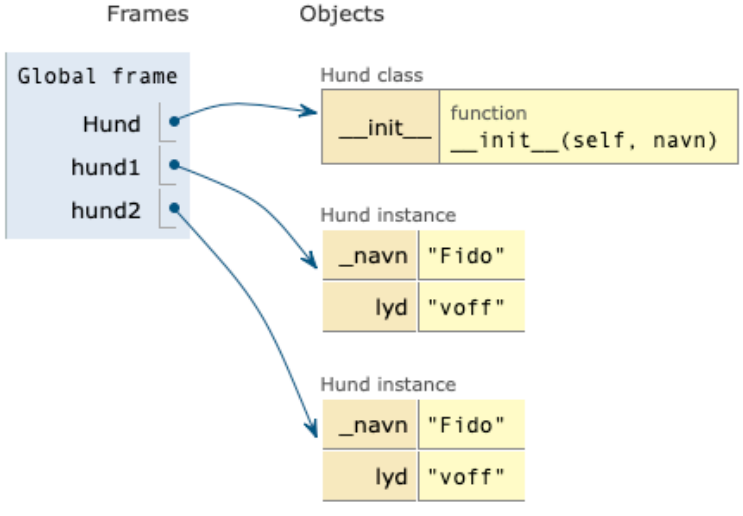
Men for objekter av egne klasser fungerer det ikke

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  class Hund:
2      def __init__(self, navn):
3          self._navn = navn
4          self.lyd = "voff"
5
6  hund1 = Hund("Fido")
7  hund2 = Hund("Fido")
8
9  print(hund1 == hund2)
```

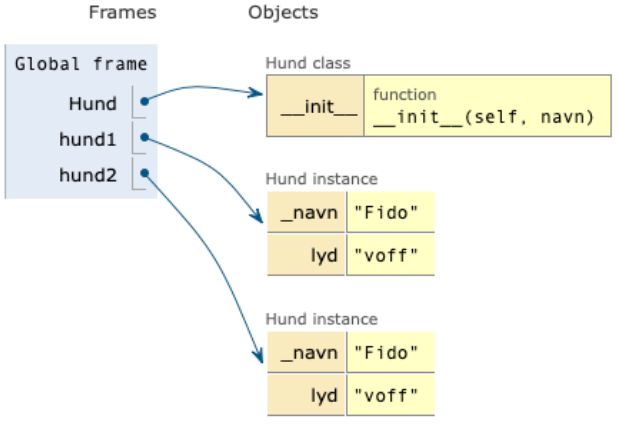
Print output (drag lower right corner to resize)

```
False
```



Selv om hundene heter det samme..

- peker referansene på forskjellige objekter
- Derfor er
  hund1 == hund2
  ikke sant




```
print(hund1 != hund2)
```

Print output (drag lower right corner to resize)

```
True
```

Men vi kan definere en metode for å sammenligne objekter

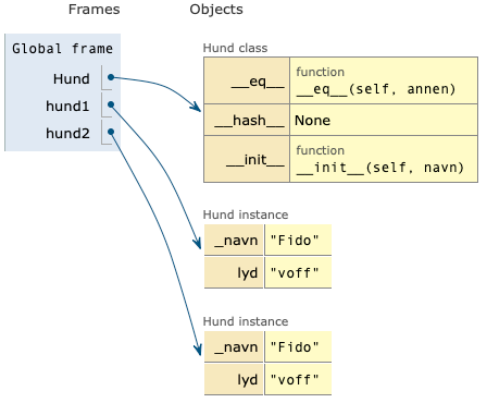
- Den magiske metoden `__eq__(self)`

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  class Hund:
2      def __init__(self, navn):
3          self._navn = navn
4          self.lyd = "voff"
5
6      def __eq__(self, annen):
7          return self._navn == annen._navn
8
9  hund1 = Hund("Fido")
10 hund2 = Hund("Fido")
11
12 print(hund1 == hund2)
13
14
15
```

Print output (drag lower right corner to resize)

```
True
```



Med `__eq__(self, annen)` definerer vi hvordan objektet skal sammenlignes med et annet objekt

```
class Hund:
    def __init__(self, navn, alder):
        self._navn = navn
        self._lyd = "voff"
        self._alder = alder

    def __eq__(self, annen):
        likAlder = self._alder == annen._alder
        liktNavn = self._navn == annen._navn
        return likAlder and liktNavn
```

vars(objekt) viser alle instans-variablene til objektet

```
print( vars(hund1) )
```

```
{'_navn': 'Fido', '_lyd': 'voff', '_alder': 21}
```

```
class Hund:
    def __init__(self, navn, alder):
        self._navn = navn
        self._lyd = "voff"
        self._alder = alder
```

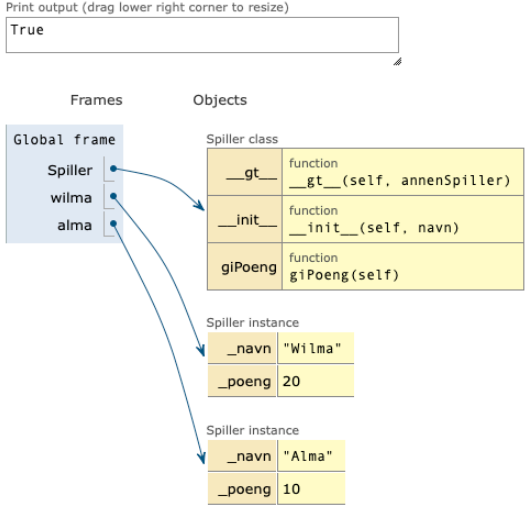
dir(objekt) viser alle metodene til objektet

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__return__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', '_alder', '_lyd', '_navn', 'getNavn']

En klasse kan implementere de magiske metodene vi trenger

- Kanskje vi skal sjekke hvem som har flest poeng?

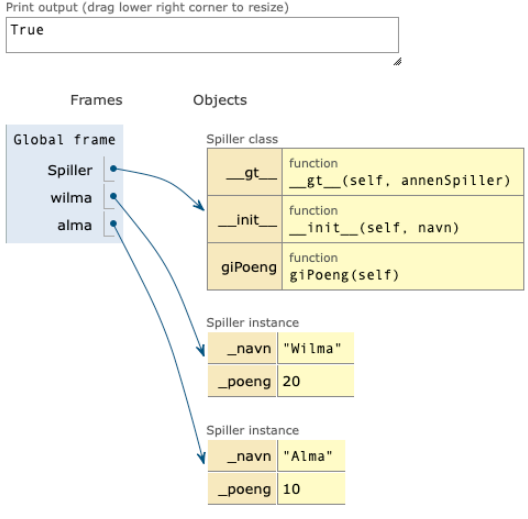
```
1  class Spiller:
2      def __init__(self, navn):
3          self._navn = navn
4          self._poeng = 0
5
6      def giPoeng(self):
7          self._poeng += 10
8
9      def __gt__(self, annenSpiller):
10         return self._poeng > annenSpiller._poeng
11
12 wilma = Spiller("Wilma")
13 alma = Spiller("Alma")
14
15 wilma.giPoeng()
16 wilma.giPoeng()
17 alma.giPoeng()
18
19 print(wilma > alma)
20
```

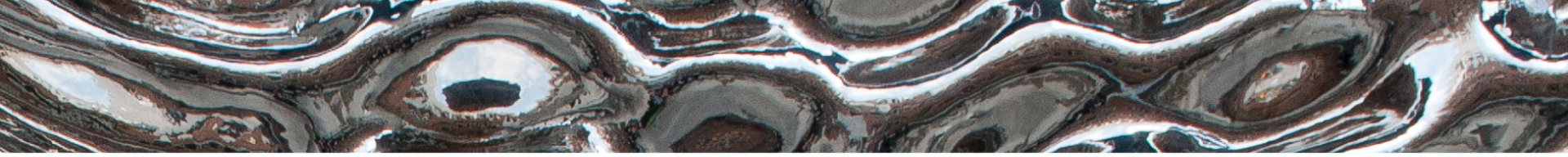


gt betyr greater than

Metoden kjører når vi sammenligner med >

```
1  class Spiller:
2      def __init__(self, navn):
3          self._navn = navn
4          self._poeng = 0
5
6      def giPoeng(self):
7          self._poeng += 10
8
9      def __gt__(self, annenSpiller):
10         return self._poeng > annenSpiller._poeng
11
12 wilma = Spiller("Wilma")
13 alma = Spiller("Alma")
14
15 wilma.giPoeng()
16 wilma.giPoeng()
17 alma.giPoeng()
18
19 print(wilma > alma)
20
```





UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Samlinger med objekter

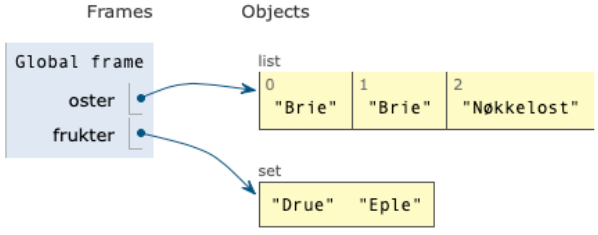
Lister, ordbøker og set

- Vi har sett på tre typer samlinger
  - list
  - dictionary
  - set

list og set

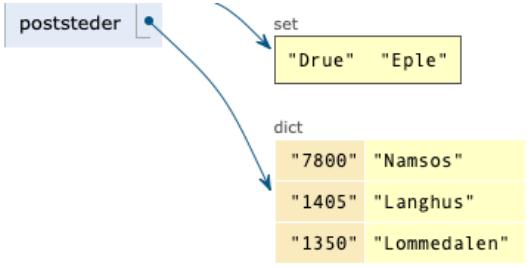
Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  oster = ["Brie", "Brie", "Nøkkelost"] # Kan ha duplikater
2
3  frukter = {"Drue", "Drue", "Eple"} # Ingen duplikater
4
5
6
7
8
```



dictionary

```
poststeder = {
    "7800" : "Namsos",
    "1405" : "Langhus",
    "1350" : "Lommedalen"
}
```



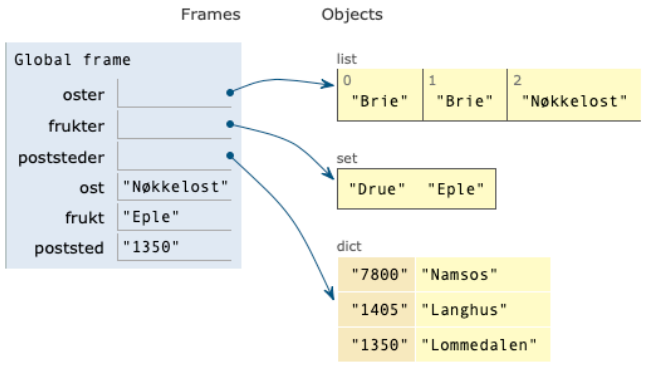
iterere over samlinger

Write code in Python 3.6 (drag lower right corner to resize code editor)

```
1  oster = ["Brie", "Brie", "Nøkkelost"] # Kan ha duplikater
2
3  frukter = {"Drue", "Drue", "Eple"} # Ingen duplikater
4
5
6  poststeder = {
7      "7800" : "Namsos",
8      "1405" : "Langhus",
9      "1350" : "Lommedalen"
10 }
11
12 for ost in oster:
13     print(ost)
14
15 for frukt in frukter:
16     print(frukt)
17
18 for poststed in poststeder:
19     print(poststed)
20
21
22
23
```

Print output (drag lower right corner to resize)

```
Brie
Brie
Nøkkelost
Drue
Eple
7800
1405
1350
```



Liste med egendefinerte objekter

- Vi skal lage et program for å velge informatikk-emner
- Krav: Skal kunne liste ut alle emner med id(emnekode), antall poeng og om kurset går på høsten eller om våren
- Designer en klasse Emne
- Legger alle emnene i en liste

Klassen Emne

kode.py emne.py

emne.py > ...

```python
class Emne:
    def __init__(self, emnekode, semester, studiepoeng):
        self._emnekode = emnekode
        self._semester = semester
        self._studiepoeng = studiepoeng

    def __str__(self):
        tekst = f"{self._emnekode} ({self._semester}) : {self._studiepoeng} studiepoeng"
        return tekst
```

En liste med objekter av klassen Emne

kode.py ● emne.py

kode.py > ...

```python
from emne import Emne

emner = []

in1000 = Emne("IN1000", "Vår", 10)
in1010 = Emne("IN1010", "Vår", 10)
in2090 = Emne("IN2090", "Høst", 10)

emner.append(in1000)
emner.append(in1010)
emner.append(in2090)

for emne in emner:
    print(emne)
```

Hva om vi skal søke etter et bestemt emne?

Husk prinsippet om innkapsling

emne.py > ...

```python
class Emne:
    def __init__(self, emnekode, semester, studiepoeng):
        self._emnekode = emnekode
        self._semester = semester
        self._studiepoeng = studiepoeng

    def __str__(self):
        tekst = f"{self._emnekode} ({self._semester}) : {self._studiepoeng} studiepoeng"
        return tekst

    def getEmnekode(self):
        return self._emnekode

```

Hva om vi skal søke etter et bestemt emne?

Vi aksesserer emnekoden med en metode i klassen

kode.py > ...

```python
from emne import Emne

emner = []

in1000 = Emne("IN1000", "Vår", 10)
in1010 = Emne("IN1010", "Vår", 10)
in2090 = Emne("IN2090", "Høst", 10)

emner.append(in1000)
emner.append(in1010)
emner.append(in2090)

query = "IN1000"

for emne in emner:
    if emne.getEmnekode() == query:
        print(emne)
```

Hva om vi ønsker å vite hvilke kurs som går om våren?

```python
class Emne:
    def __init__(self, emnekode, semester, studiepoeng):
        self._emnekode = emnekode
        self._semester = semester
        self._studiepoeng = studiepoeng

    def __str__(self):
        tekst = f"{self._emnekode} ({self._semester}) : {self._studiepoeng} studiepoeng"
        return tekst

    def getEmnekode(self):
        return self._emnekode

    def getSemester(self):
        return self._semester
```

```python
query = "Vår"

for emne in emner:
    if emne.getSemester() == query:
        print(emne)
```

Vi kan kanskje la folk skrive inn semester selv

```
query = input("Velg semester (Vår eller Høst): ")

for emne in emner:
    if emne.getSemester() == query:
        print(emne)
```

```
Velg semester (Vår eller Høst): Vår
IN1000 (Vår) : 10 studiepoeng
IN1010 (Vår) : 10 studiepoeng
```

Alternativt – Lagre emner i en ordbok

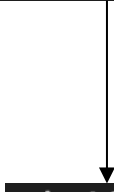
- Legg merke til at vi må skrive `emner.values()` for å få ut verdiene til objektene

kode.py > ...

```python
from emne import Emne

emner = {}

in1000 = Emne("IN1000", "Vår", 10)
in1010 = Emne("IN1010", "Vår", 10)
in2090 = Emne("IN2090", "Høst", 10)

emner["in1000"] = in1000
emner["in1010"] = in1010
emner["in2090"] = in2090

for emne in emner.values():
    print(emne)
```

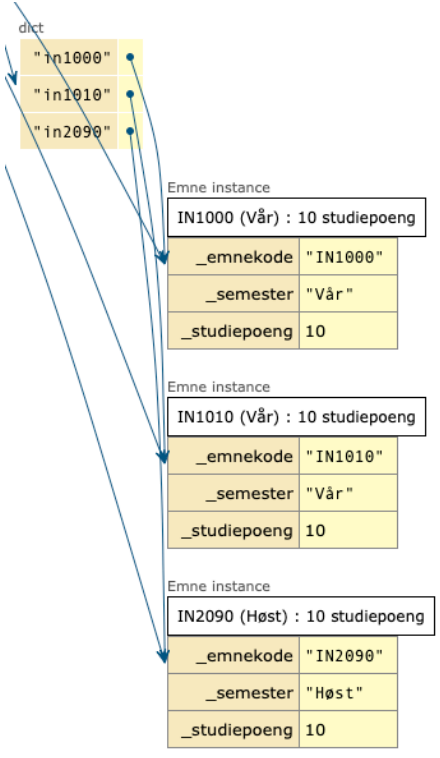
For-løkken vil ellers iterere over nøklene

```
for emne in emner:
    print(emne)
```



```
in1000
in1010
in2090
```

Ordboken inneholder peker til objektene



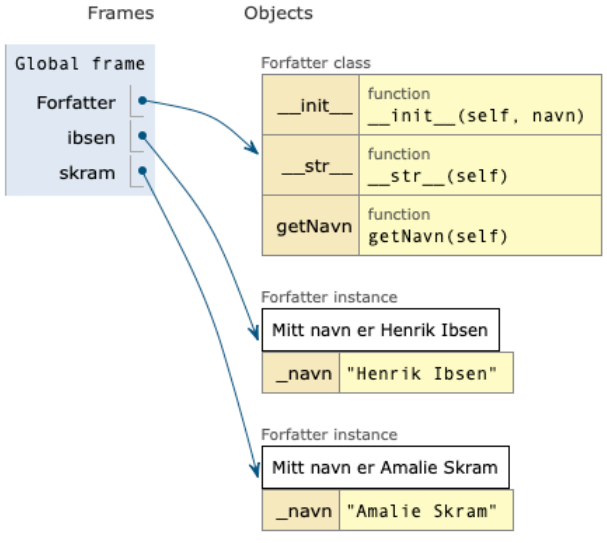
Objekter med referanser til andre objekter

- Vi skal nå opprette to klasser
  - Forfatter
  - Bok
- En bok er skrevet av en forfatter

Først lager vi klassen Forfatter

Write code in Python 3.6 (drag lower right corner to resize code editor)

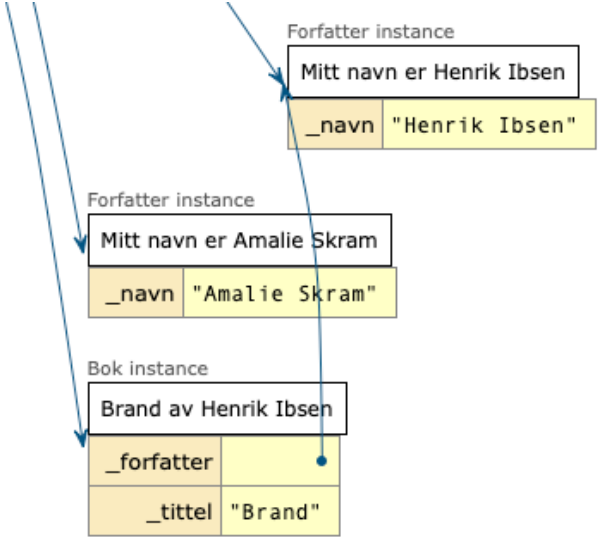
```
1  class Forfatter:
2      def __init__(self, navn):
3          self._navn = navn
4
5      def __str__(self):
6          return f"Mitt navn er {self._navn}"
7
8      def getNavn(self):
9          return self._navn
10
11
12
13 ibsen = Forfatter("Henrik Ibsen")
14 skram = Forfatter("Amalie Skram")
15
16
17
```



Deretter lager vi klassen Bok, som har en referanse til et objekt av klassen Forfatter

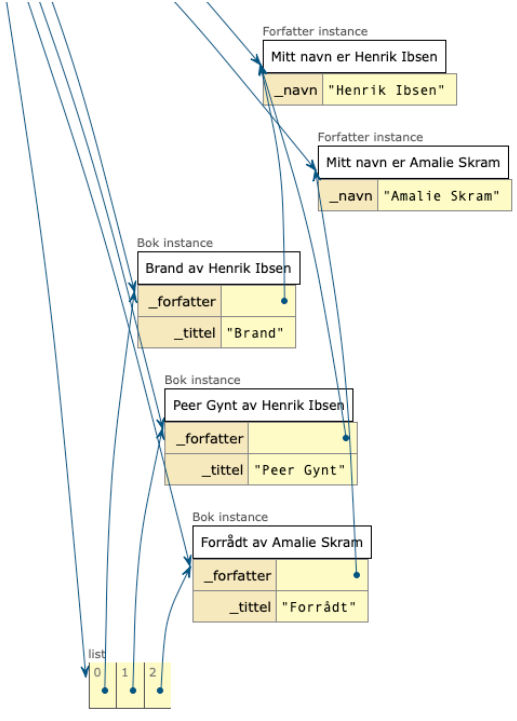
```
class Bok:
    def __init__(self, tittel, forfatter):
        self._tittel = tittel
        self._forfatter = forfatter

    def __str__(self):
        return self._tittel + " av " + self._forfatter.getNavn()


ibsen = Forfatter("Henrik Ibsen")
skram = Forfatter("Amalie Skram")

brand = Bok("Brand", ibsen)
```

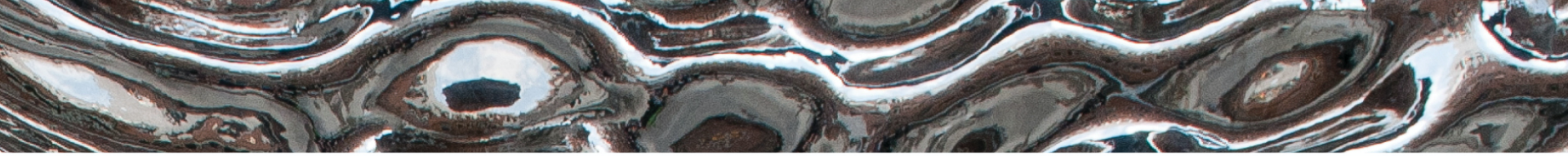
I objektet brand er det en referanse til forfatteren ibsen



Vi kan opprette mange bok-objekter og legge dem i en liste

```
ibsen = Forfatter("Henrik Ibsen")
skram = Forfatter("Amalie Skram")

brand = Bok("Brand", ibsen)
peer = Bok("Peer Gynt", ibsen)
forradt = Bok("Forrådt", skram)

boker = [brand, peer, forradt]
```

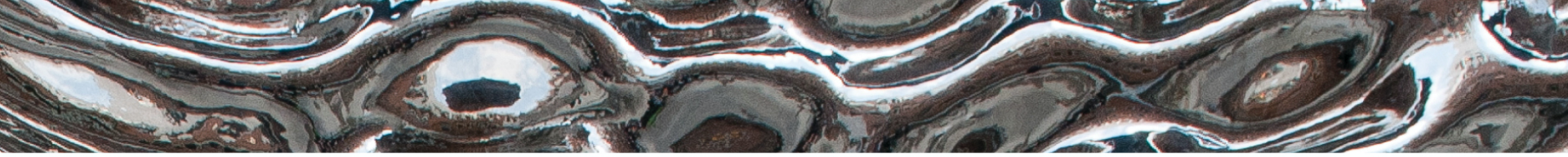




UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Plan for dagen

1. **Repetisjon om innkapsling**
2. **Magiske metoder**
3. **Samlinger av objekter**



UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Repetisjon om innkapsling

Klasse og grensesnitt

- Klassen er mønsteret eller modellen vi lager objekter etter
- Klassedefinisjonen bestemmer hvilke data objektene kan lagre (instansvariabler) og hva objektene kan gjøre (instansmetoder)
- Grensesnittet er de metodene vi tilbyr programmereren
- Ingen variabler skal være tilgjengelige fra utsiden – vi markerer dette ved å sette _ foran variabelnavnet
- _ vil si at instansvariablene er non-public

Innkapsling

- Innkapsling er som en brus-automat
- Du trykker på en knapp fra utsiden
  - og ut kommer en brus (returverdi)
- Vi ønsker ikke at folk skal stikke armen inn i brusautomaten og hente ut en brus

Enkle og sammensatte data-typer

- Vi har brukt variabler av ulike data-typer
  - heltall (1, 43, -20)
  - boolske verdier (true, false)
- Og vi har brukt mer sammensatte data-typer
  - strenger
  - lister
  - ordbøker

Enkle og sammensatte data-typer

- Vi har brukt variabler av ulike data-typer
  - heltall (1, 43, -20)
  - boolske verdier (true, false)
- Og vi har brukt mer sammensatte data-typer
  - strenger
  - lister
  - ordbøker

Eksempel på datatyper med tjenester

- Lister
  - legge til med append(element)
  - ta ut med pop()
  - ta ut med pop(indeks)
- Tekster
  - fjerne blanke med strip()
  - små bokstaver med lower()

Vi kan lage våre egne typer (datatyper)

- Ved å definere våre egne klasser kan vi selv "konstruere" slike sammensatte, men skreddersydde typer
- Deretter kan vi opprette og bruke ett eller flere objekter av hver klasse
- Dette er nyttig i mange sammenhenger

For eksempel for en konkurranse der det er om å gjøre å spise flest bananer

- Kode i python-tutor

Klassen Bananspiser

```
kalle.spisBanan()
```

Vi ber om en endring utenfra

Vi henter ut en verdi

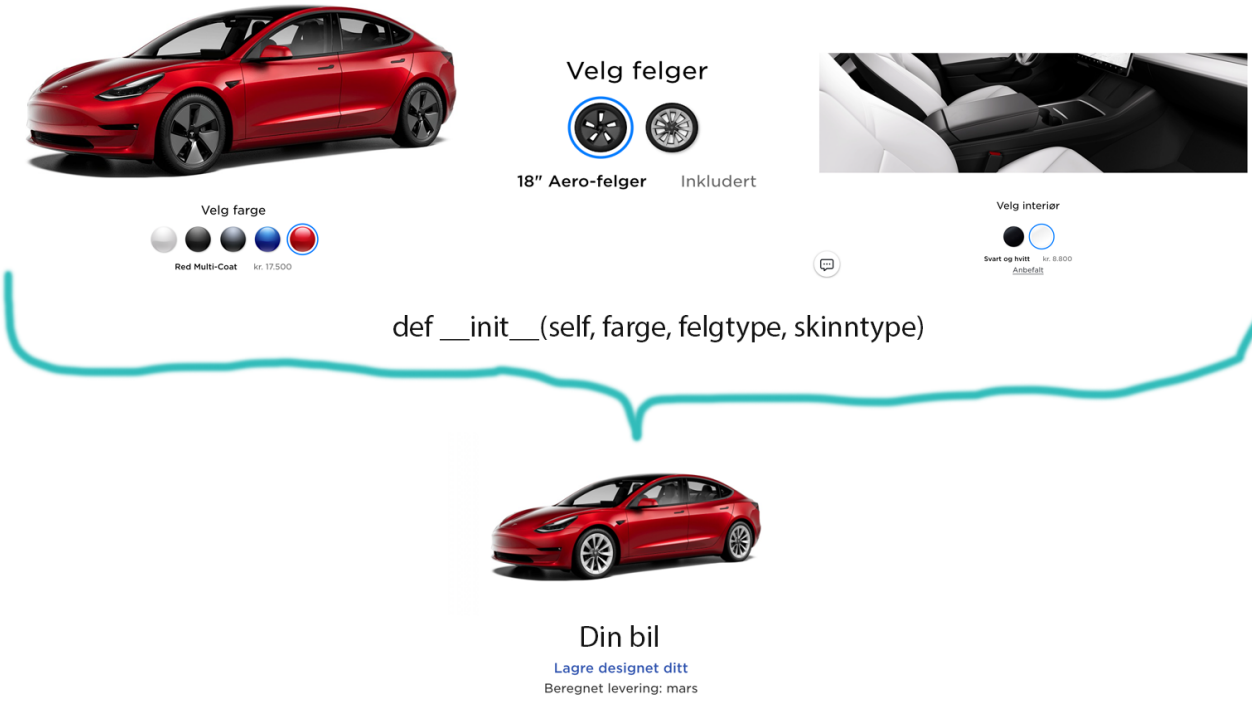
```
kalle.hentAntallBananer()
```

```
class Bananspiser:
    def __init__(self, registrertNavn):
        self._navn = registrertNavn
        self._antallBananerSpist = 0

    def spisBanan(self):
        self._antallBananerSpist += 1

    def hentAntallBananer(self):
        return self._antallBananerSpist

kalle = Bananspiser("Kalle")
stine = Bananspiser("Stine")

kalle.spisBanan()
stine.spisBanan()

print( kalle.hentAntallBananer() )
```

Så hva er en klasse?

- Et mønster / en modell
- En arkitekt-tegning som beskriver hvordan et hus skal bygges
- En pepperkakeform?
  - Før vi begynner å bake har vi ingen pepperkaker, men vi har formen
  - Vi kan lage så mange pepperkaker vi vil med pepperkakeformen

Men alle objekter av en klasse er ikke helt like



Konstruktøren

- **Klassen** har ingen verdier – den definerer kun hvilke verdier **objektene** skal ha
- Det er konstruktøren som setter verdiene til **objektene** av **klassen**
  - Vi kaller gjerne disse verdiene for **attributter**

Konstruktørens oppgave

- Konstruktøren setter verdier

```
class Bil:
    def __init__(self, farge, skinntype, felgtype):
        self._kmStand = 0
        self._hestekrefter = 460
        self._farge = farge
        self._skinntype = skinntype
        self._felgtype = felgtype
```

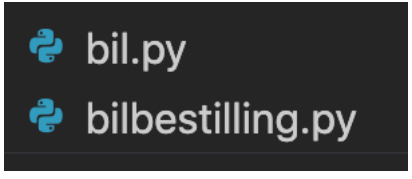
Noen verdier bestemmer vi når vi oppretter objekter av klassen

Andre verdier er pre-definerte i klassen

(Gjen)bruk av klasser

- Vi lagrer gjerne hele koden for klassen i en egen fil. Filnavnet kan være det samme som klassenavnet, men med liten forbokstav.

- Vi importerer klassen i **andre** python-filer og oppretter objekter




```python
from bil import Bil

def hovedprogram():
    bil1 = Bil("rød", "sort nappa", "speed-felger")
    bil2 = Bil("hvit", "hvit imitert", "aero-felger")
```

Hvorfor lage en klasse?

- For å tilby et sett tjenester som hører sammen (verktøykasse)
- Spesielt nyttig for tjenester som trenger å dele data over tid (eks for håndtering av en liste, eller en fil, eller en tekst)
- For å kunne lage flere like objekter med samme data og oppførsel (eks representere studenter)
- For å simulere sammensatte fenomener vha objekter som representerer virkelige elementer (eks trafikksimuleringer)