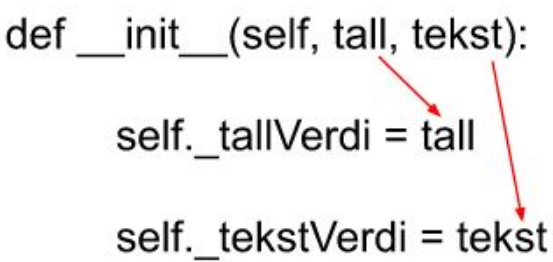
Gruppetime 9

Mer om klasser, magiske metoder

Ukens viktigste saker

- Mer objektorientering
- Mer komplekse datastrukturer
- Magiske metoder
- Repetisjon

Litt repetisjon



Variabler med konstant startverdi: trenger **ikke** å være parameter.

```
def __init__(self, tall, tekst):
      self._tallVerdi = tall
      self._tekstVerdi = tekst
      self._startVerdi = 0
```

Oppgave 1

Hvilke verdier får variablene til instansene av objektene av type Person her, gitt opprettelsen av person slik: test_person = Person(13, “Kari”)

```
def __init__(self, alder, navn):
    self._alder = 0
    self._navn = “”
```

```
def __init__(self, alder, navn):
    self._alder = navn
    self._navn = alder
```

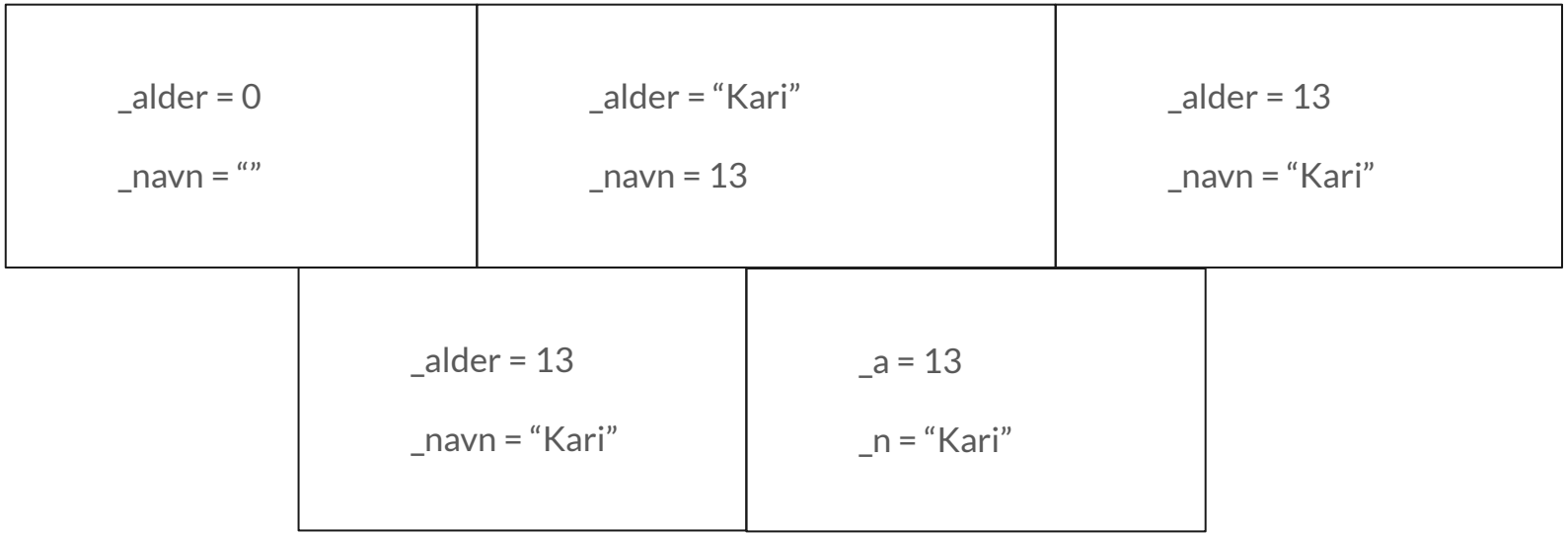
```
def __init__(self, alder, navn):
    self._alder = alder
    self._navn = navn
```

```
def __init__(self, a, b):
    self._alder = a
    self._navn = b
```

```
def __init__(self, alder, navn):
    self._a = alder
    self._n = navn
```

Oppgave 1 - Løsningsforslag

Hvilke verdier får variablene til instansene av objektene av type Person her, gitt opprettelsen av person slik: `test_person = Person(13, “Kari”)`



Oppgave 2

Gitt følgende program, opprett 3 instanser av hver med fornuftige verdier.

dersom metthet alltid skal settes til 10 ved start trenger ikke metthet å være med som parameter i konstruktøren

```
class Bil:
    def __init__(self, regnr, type, aars_modell):
        self._regnr = regnr
        self._type = type
        self._aars_modell = aars_modell
```

```
class Koffert:
    def __init__(self, toalettsaker_liste, kles_liste):
        self._toalettsaker = toalettsaker_liste
        self._klær = kles_liste
```

```
class Hund:
    def __init__(self, a, b, c):
        self._alder = a
        self._rase = b
        self._bjeffer_mye = c
        self._metthet = 10
```

Oppgave 2 - Løsningsforslag

Gitt følgende program, opprett 3 instanser av hver med fornuftige verdier.

bil1 = Bil(“AB123”, “Toyota”, 1997) bil2 = Bil(“EL456”, “BMW”, 2016) bil3 = Bil(“DB5435”, “Citroen”, 2004)	bag1 = Koffert([“tannbørste”, “tannkrem”], [“jakke”, “genser”, “bukse”]) toalettS = [“hårbørste”, “neglsaks”, “tannbørste”] klær = [“sokker”, “tskjorte”, “shorts”] bag2 = Koffert(toalettS, klær) bag3 = Koffert(toalettS, [“badetøy”, “skjorte”, “jeans”])	labrador = Hund(10, “Labrador”, False) shiba = Hund(4, “Shiba Inu”, True) dachs = Hund(7, “Dachs”, True)



Metoder - når trenger man return?

Metoder - når trenger man return?

- Metoder trenger ikke å returnere noe!
- Return og print() har ikke noe med hverandre å gjøre
- Return er nyttig når vi vil hente en verdi
- print() er nyttig når vi skal skrive ut noe direkte til skjerm
- Les obligen nøye: Står det at dere skal returnere svaret skal dere bruke return. Står det at dere skal skrive ut skal dere bruke print()

Metoder - når trenger man return?

- Noen huskeregler:
  - metoder som heter settInstansvariabel() eller lignende skal ofte ikke returnere noe
  - metoder som heter hentInstansvariabel() eller lignende skal ofte returnere noe
  - (disse metodene er det vanlig å kalle getters og setters)
  - metoder som heter skrivInfo() eller lignende bruker ofte print(), men skal gjerne *ikke* returnere noe
  - metoder som heter hentStreng(), __str__(), __repr__() og lignende skal returnere en streng, men ikke skrive ut noe
  - Hvis en metode vil gjøre to av følgende: print(), return, endre instansvariabel - tenk deg om to ganger (men ikke ulovlig). Kanskje bedre med separate metoder?

Return i løkker

```
def liste_har_negativt_tall(liste):
    for tall in liste:
        if tall < 0:
            return True
        else:
            return False
```

Hva skjer her?

Hva er feil med denne koden?

- gå stegvis gjennom et eksempel på en liste

Magiske metoder

Magiske metoder er spesielle metoder i Python som vi kan overskrive for å bruke bestemte operatorer.

- Markeres med to understreker __ før og etter navnet på metoden
  - Eksempel som vi allerede har sett: __init__
- Vi skal aldri *kalle* på metoden *direkte med navnet*, f.eks. **ikke:** obj.__str__()

__str__

Drøft forskjellen mellom hentStreng og __str__ metoden her.

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

        def __str__(self):
                return self._navn

         def hentStreng(self):
                print(self._navn)

nyHund = Hund(“Ole”, 19)
nyHund.hentStreng()
print(nyhund)


if nyHund.hentStreng() == “Ola”:
    print(“Ola Normann?”)
if str(nyHund) == “Ola”:
    print(“Ola Nordmann?)
```

__str__

Implementering av denne gjør at vi enkelt kan representere objektet vårt som en streng. Det er vi som definerer hvordan denne brukervennlige utskriften skal se ut.

- Kalles når vi bruker str(obj) eller print(obj)

Eksempel fra forelesning:

```
class Rektangel:
        def __init__(self, len, bredde):
                self._lengde = len
                self._bredde = bredde

def __str__(self):
        pen_streng = “Lengde “ + self._lengde + “, bredde: “ + self._bredde
return pen_streng
```

```
r1 = Rektangel(8, 6)
print(r1) -> Lengde: 8, bredde: 6
```

__eq__()

__eq__ :

- gjør det mulig å sammenlikne objekter med **==** basert på det *vi* definerer til å være "*like objekter*".
- kalles automatisk på når vi bruker == mellom to objekter av en klasse med denne metoden definert

```
class Rektangel :
   def __init__(self, lengde, bredde):
      self._lengde = lengde
      self._bredde = bredde

   def __eq__(self, annen):
      return (self._lengde == annen._lengde and self._bredde ==
annen._bredde)

r1 = Rektangel(8,6)
r2 = Rektangel(8,6)
print (r1 == r2)   -> True

r3 = Rektangel(6,4)
print (r2 == r3)   -> False
```

Sammenlikning: is vs. ==

`X is Y` : Er det samme objekt (lik adresse)?

`X == Y` : Er det like objekt (likt innhold)?

Eksempler:

a = 5

b = a

b is a → False

b == a → True

person1 = Person(“Per”)
person2 = person1

person1 is person2 → True
person1 == person2 → True

---

person1 = Person(“Per”)
person2 = Person(“Per”)

person1 is person2 → False
person1 == person2 → True

Sjekke om noe er None

- bruk **is** i stedet for **==**
- Grunnen til dette er at None er ett objekt - vi kan ikke ha to instanser av None
- **is** gir derfor alltid riktig svar
- **==** er derimot avhengig av hvordan en klasse har implementert __eq__()

```
person1 = Person(“Per”)
person2 = None

person1 is None # False
person2 is None # True
person1 == None # Avhengig av implementasjon
                # av __eq__() i klassen
```

Oblig 7

- Skriv egne testprogrammet og test med utdelte tester til slutt
- Hvordan leser man test-utskriften?
- Spørsmål?