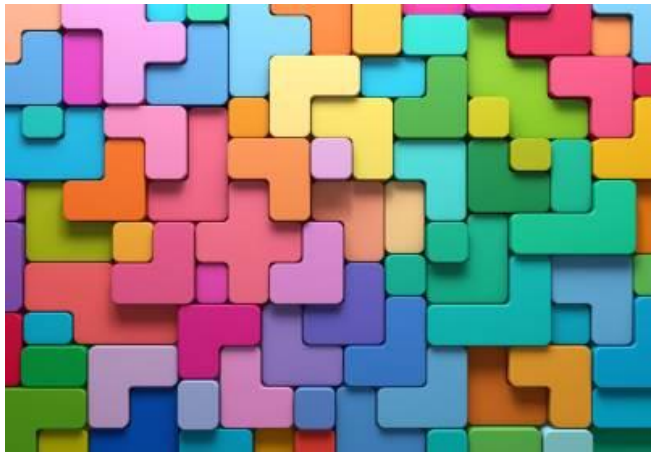
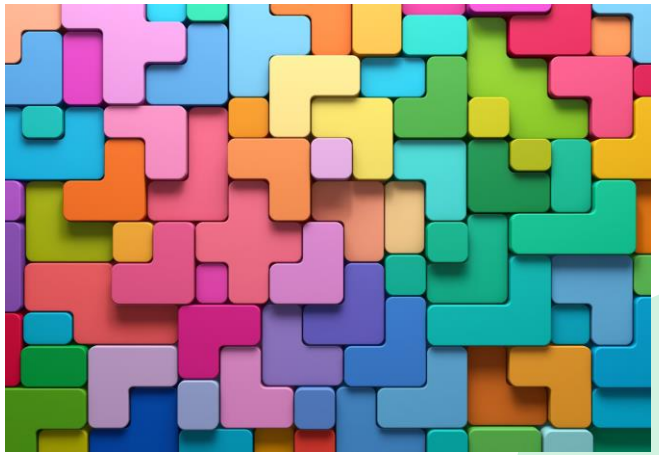
ARV

FORELESNING 7

ONSDAG 11/9





(bilder generert av bing image creator)

- https://www.menti.com

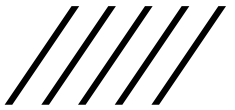
8522 9533

En artikkel om læring

Despite active learning being recognized as a superior method of instruction in the classroom, a major recent survey found that most college STEM instructors still choose traditional teaching methods. This article addresses the long-standing question of why students and faculty remain resistant to active learning. Comparing passive lectures with active learning using a randomized experimental approach and identical course materials, we find that students in the active classroom learn more, but they feel like they learn less. We show that this negative correlation is caused in part by the increased cognitive effort required during active learning. Faculty who adopt active learning are encouraged to intervene and address this misperception, and we describe a successful example of such an intervention.

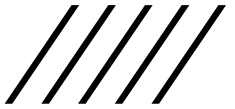
Læremål: Avansert bruk av Python

- Gjøre programmene lettere å lese og finne fram i
- *Objektorientert programmering* er en måte å tenke på når vi skriver programmer som åpner nye muligheter
- Lar oss lage spesialtilfeller fra mer generelle tilfeller (arv)
- (+ mer de kommende ukene)

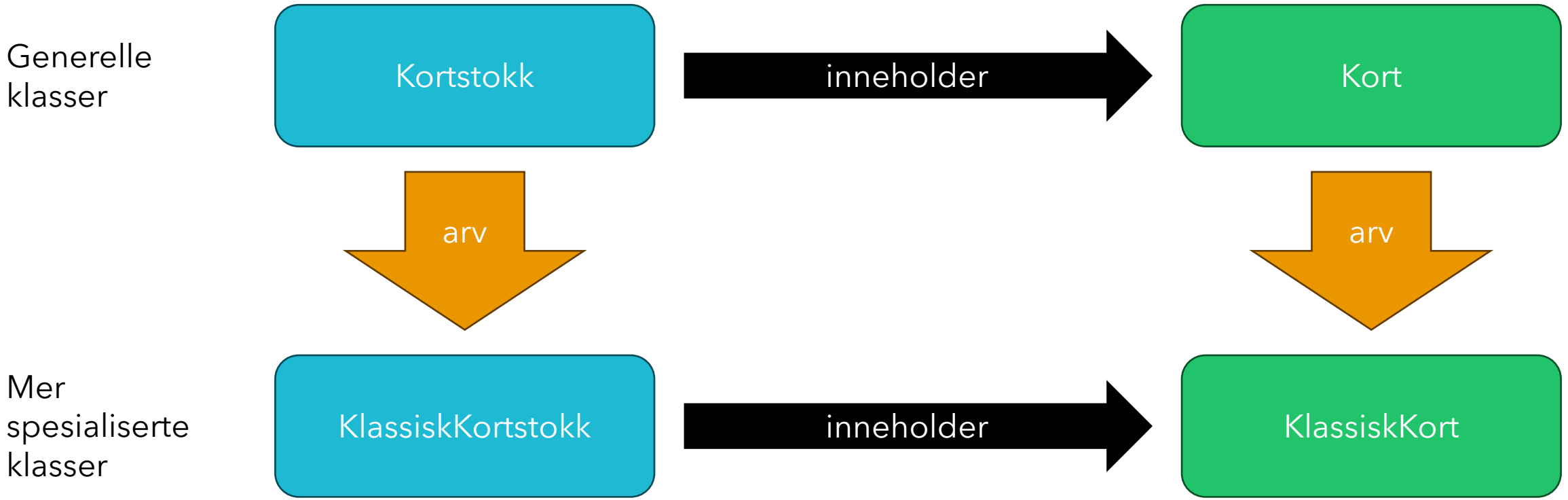


Livekoding over flere forlesninger

- Fortsett der vi slapp i forelesning 5
- Hvis du ikke var til stede eller ikke finner koden:
  - Lag en fork av dette repo'et (public)
  - Direkte link: https://github.uio.no/oddps/IN1910-live/fork



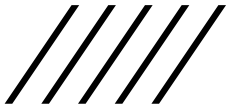
Klassediagram

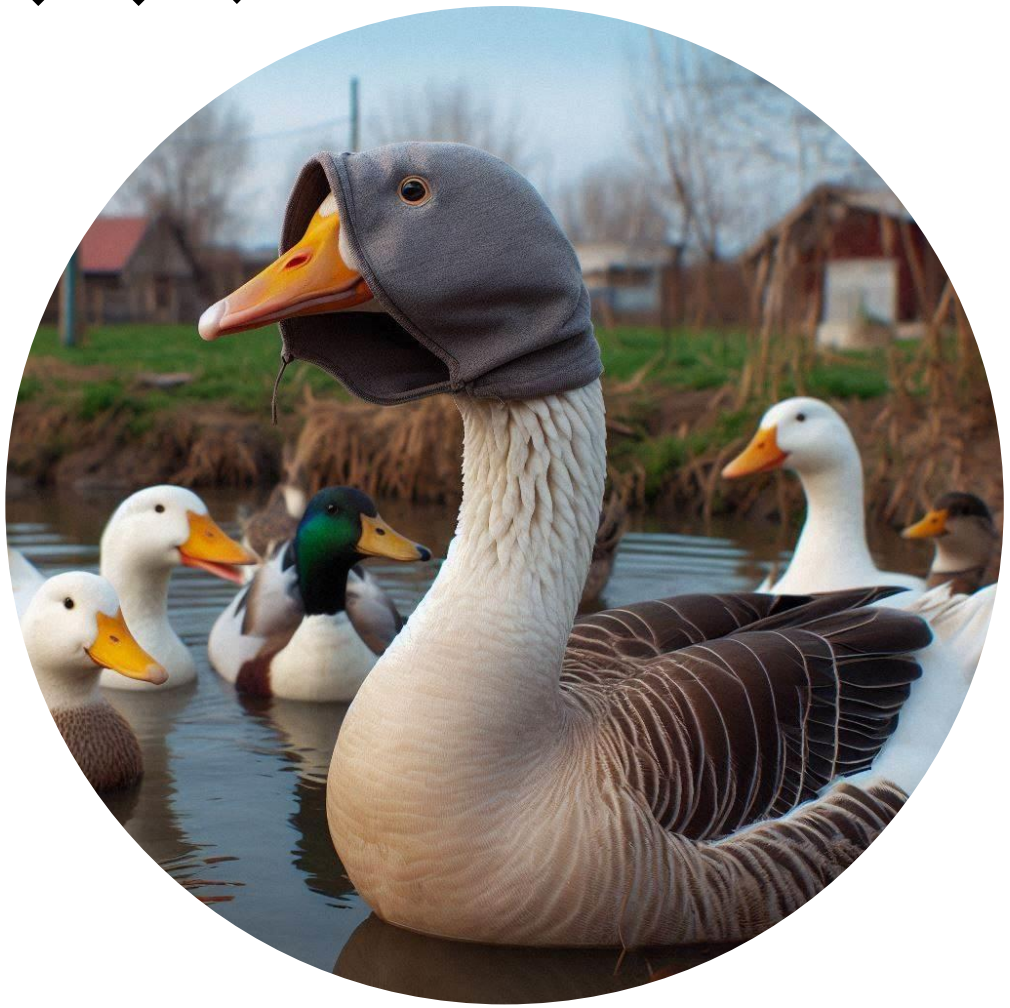


LIVEKODING: KLASSER

Bør vi alltid bruke arv?

- Nei, ikke alltid!
- Arv *kan* være fornuftig hvis
  - Vi vil ha en ny klasse som gjør mye av det en annen klasse gjør + noe ekstra (slipper å copy-paste kode)
  - Du har flere klasser som gjør akkurat det samme, og vil oppdatere dem *ett sted* (i moderklassen) i stedet for å oppdatere dem hver for seg (**Goose** er oppdatert, men du glemte **Duck** og **Swan**)
  - Du bruker type-sjekking og synes **b: Bird** er mer oversiktelig enn **b: Union[Duck, Goose, Albatross]**





Hva er alternativet?

- *Duck typing* (Python):
  - “if it **.swim()** like a **Duck** and **.fly()** like a **Duck**, it must be a **Duck**”
- alle klasser med de rette metodene gjør at koden kjører uten feil, selv om det ikke er “riktig” klasse
- (men **mypy** gjennomskuer det)

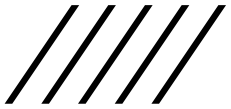
```python
class Duck:
    def swim(self) -> None:
        print("The duck is swimming.")

    def fly(self) -> None:
        print("The duck is flying.")

class Goose:
    def swim(self) -> None:
        print("The goose is swimming.")

    def fly(self) -> None:
        print("The goose is flying.")

class Albatross:
    def swim(self) -> None:
        print("The albatross is swimming.")

    def fly(self) -> None:
        print("The albatross is flying.")
```

Duck typing (uten arv)

```python
birds:list[Duck] = [Duck(), Goose(), Albatross()]

for bird in birds:
    bird.fly()
    bird.swim()
```

```
The duck is flying.
The duck is swimming.
The goose is flying.
The goose is swimming.
The albatross is flying.
The albatross is swimming.
```

```
(base) PS C:\GitHub\IN1910-live\forelesning 5> mypy ducks.py
ducks.py:23: error: List item 1 has incompatible type "Goose"; expected "Duck"  [list-item]
ducks.py:23: error: List item 2 has incompatible type "Albatross"; expected "Duck"  [list-item]
Found 2 errors in 1 file (checked 1 source file)
```



Samme eksempel med arv

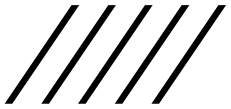
```python
class Bird:
    def swim(self) -> None:
        print(f"The {type(self).__name__.lower()} is swimming.")

    def fly(self) -> None:
        print(f"The {type(self).__name__.lower()} is flying.")
```

```python
class Duck(Bird):
    pass

class Goose(Bird):
    pass

class Albatross(Bird):
    pass
```

```python
birds: list[Bird] = [Duck(), Goose(), Albatross()]

for bird in birds:
    bird.fly()
    bird.swim()
```

```
The duck is flying.
The duck is swimming.
The goose is flying.
The goose is swimming.
The albatross is flying.
The albatross is swimming.
```

```
(base) PS C:\GitHub\IN1910-live\forelesning 5> mypy birds.py
Success: no issues found in 1 source file
```



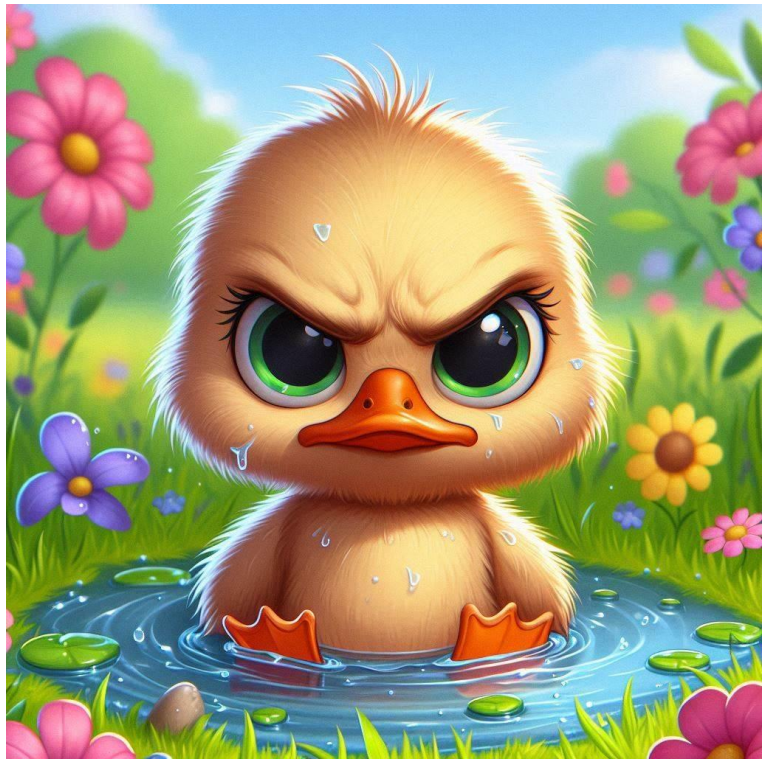

Oppgave: Hva skjer om vi *fjerner* arv fra klassene vi livekodet?

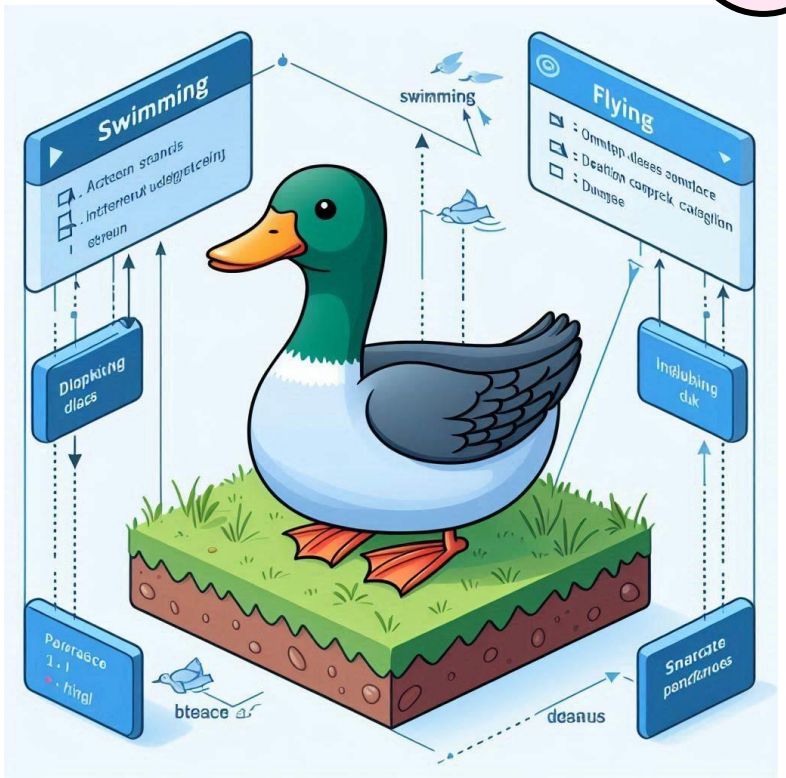
Fordeler med duck typing i Python

- Koden blir enkel og fleksibel når du kan fokusere på hva objektene *gjør* og ikke hva de *er*
- Du kan lage enkle *prototyper* av mer kompiserte klasser for å teste annen kode i en tidlig fase
- Du kan gjenbruke en klasse et annet sted uten å importere *alle* klassene som arver fra hverandre

Ulemper med duck typing i Python

- Programmet kan kræsje eller gi feilmelding når det kjøres fordi det mangler en metode eller atributt som man antok var der
- Det kan være vanskelig å vite hva slags objekt det er snakk om når man leser koden
- Det kan bli mer krevende å finne feil når typene er udefinert

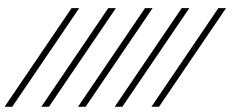




LIVEKODING: IMPLEMENTERE GRENSESNITT

Definere grensesnitt med abstract base classes (ABCs)

- En annen måte å bruke arv på
- Lager en *abstrakt klasse* = en du ikke kan lage objekter av (du kan ikke lage en **Swimming**, men du kan lage en **Duck**)
- En klasse som *arver* fra denne klassen må *implementere* alle metodene i grensesnittet
- Hvis du glemmer en får du **NotImplementedError**
- Kan dermed *tvinge* et objekt til å ha bestemte metoder fra ett (eller flere) grensesnitt



Python støtter generelt arv fra flere klasser samtidig

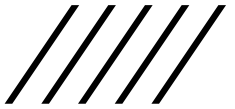
```python
class Arbeidstaker:
    def __init__(self):
        self.penger = 0

    def få_lønn(self, beløp) -> None:
        self.penger += beløp

class Student:
    def __init__(self):
        self.obliger = 0

    def lever_oblig(self) -> None:
        self.obliger += 1
```

```python
ola = Stipendiat()
ola.lever_oblig()
ola.få_lønn(30000)

print("Ola har", ola.penger, "på konto og har levert", ola.obliger, "obliger.")
```

```
Ola har 30000 på konto og har levert 1 obliger.
```

```
(base) PS C:\GitHub\IN1910-live\forelesning 7> mypy multiple.py
Success: no issues found in 1 source file
```

```python
class Stipendiat(Arbeidstaker, Student):
    def __init__(self):
        Arbeidstaker.__init__(self)
        Student.__init__(self)
```



Etter forelesningen

- Finn ditt repository for prosjekt 1
- Finn brukernavnet til din medstudent i repo-navnet (*H24_project1_<brukernavn1>_<brukernavn2>*) og ta kontakt på *<brukernavn>*@ulrik.uio.no
- (Men les først egen e-post for å se om medstudenten har kontaktet deg først.)
- Får du ikke kontakt i løpet av et par dager, eller opplever andre hindringer i prosjektarbeidet, bruk dette skjemaet