Kapittel 10: Mer om klasser og objekter

Relasjoner: arv, aggregat og komposisjon
Abstrakte klasser, interfaces og polymorfi

Relasjoner: er-en og har-en

- Vi bruker klasser for å modellere / abstrahere «ting» / entiteter fra «den virkelige verden».
- Når vi opererer med mer enn en klasse vil det gjerne være slik at det oppstår *relasjoner* mellom klassene og objektene.
- Det er spesielt to typer relasjoner som er viktige å kjenne til mellom klasser og objekter;
  - ***er-en*** relasjon (også kalt ***arv***), en relasjon mellom klasser
  - ***har-en*** relasjon, som kan være to typer: komposisjon og aggregat, en relasjon mellom objekter

Har-en (hel-del) relasjon: komposisjon

```
class Engine:
    def __init__(self, horsepower):
        self._horsepower = horsepower
class Car:
    def __init__(self, horsepower):
        self._engine = Engine(horsepower)


# Opprette en bil med en motor
car = Car(150)


# Hvis bilen slettes, slettes også motoren
del car # Motoren eksisterer ikke lenger
```

Koden til venstre viser en ***har-en*** (også kalt hel-del) relasjon av typen ***komposisjon***, som betyr:

Delene er helt avhengige av helheten.

Når helheten slettes, slettes også delene.

En klasse `Car` kan ha objekter som `Engine` og `Wheel`.

Hvis bilen slettes, gir det ikke mening at motoren eller hjulene eksisterer alene.

Har-en (hel-del) relasjon: aggregat

```
class Person:
    def __init__(self, name):
        self._name = name

class Team:
    def __init__(self):
        self._members = []
    def add_member(self, person):
        self._members.append(person)

# Opprette personer
person1 = Person("Alice")
person2 = Person("Bob")

# Opprette et team og legge til medlemmer
team = Team()
team.add_member(person1)
team.add_member(person2)
```

Aggregat er en annen, ikke så streng har-en relasjon, hvor

- Delene kan eksistere uten helheten
- Helheten og delene har en løs kobling

En klasse **`Team`** kan ha en liste med **`Person`** - objekter som medlemmer. Hvis teamet slettes, kan personene fortsatt eksistere.

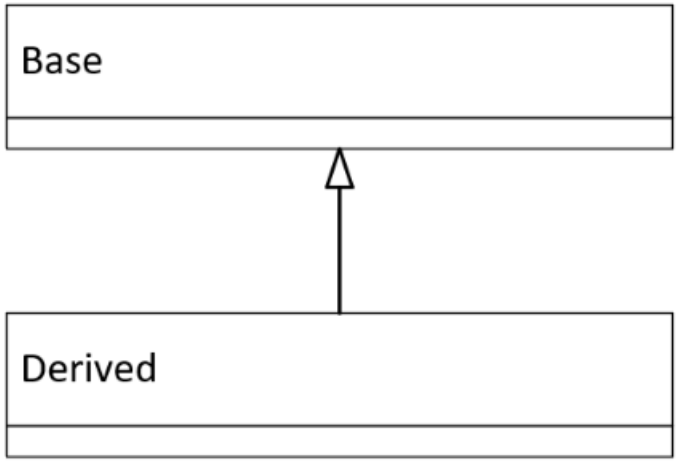
```
# Selv om teamet slettes, eksisterer
# personene fortsatt:
del team
print(person1.name)  # Output: Alice
```

Merk implementeringen av begge typer:

Uansett type ***har-en*** relasjon (komposisjon eller aggregat), så ser vi at selve relasjonen er implementert ved at det ene objektet har en referanse til (en eller flere) av den andre objekt typen.

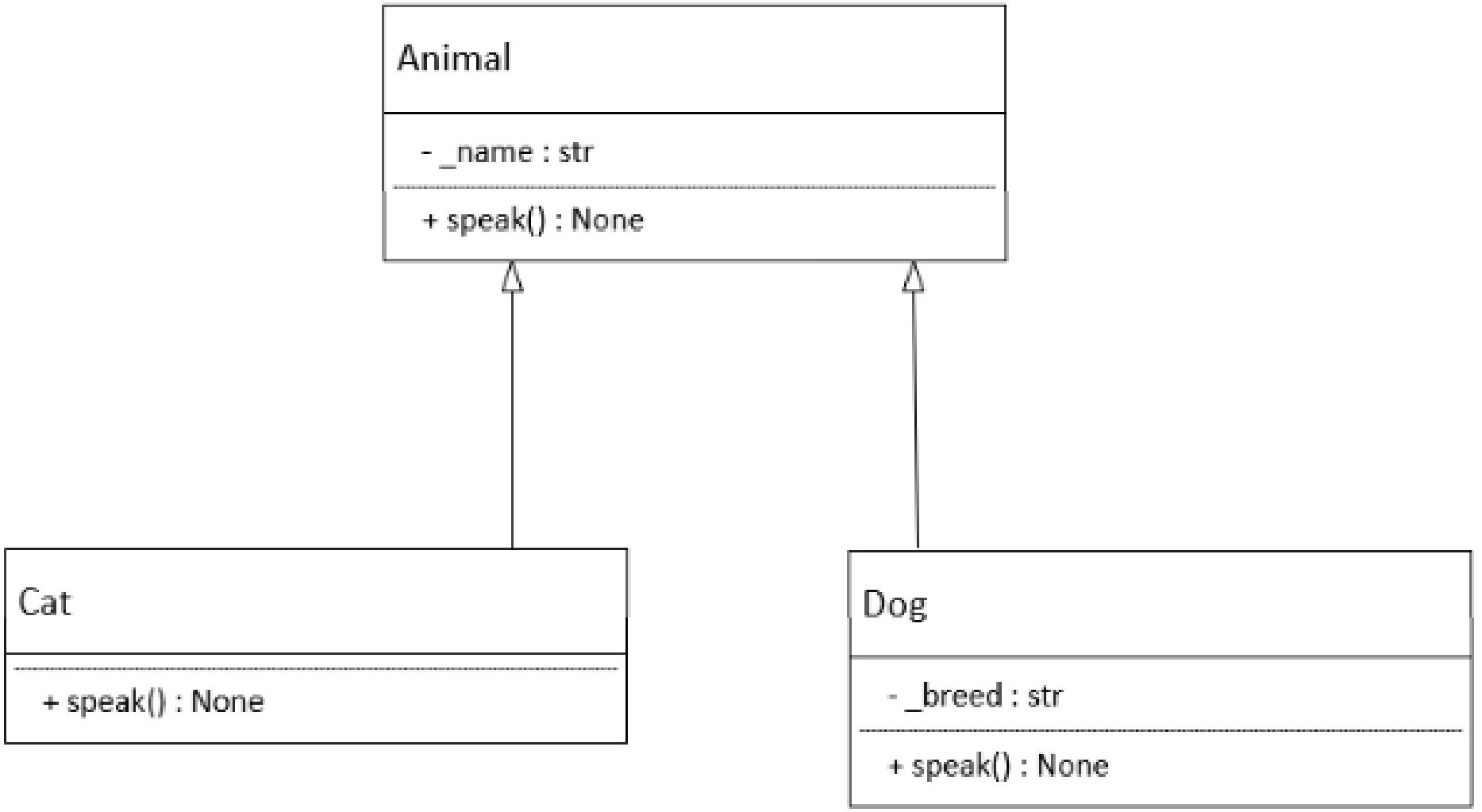
Er-en relasjon: Arv

- Arv er et grunnleggende konsept i objektorientert programmering som lar en klasse (kalt *subklasse* eller *barneklasse*) *arve* egenskaper og metoder fra en annen klasse (kalt *superklasse* eller *foreldreklasse*).
- Dette gjør det mulig å gjenbruke kode og skape hierarkier av klasser.
- Arv illustreres slik i et UML diagram (Derived arver Base)



Arv

- Arv-mekanismen lar oss bruke egenskaper (attributter) og metoder fra en annen klasse, samtidig som vi kan *legge til* eller *overstyre* funksjonalitet.
- Når en klasse arver fra en annen klasse:
  - Derived (subklassen) får tilgang til alle attributter og metoder fra base class (superklassen)
  - Subklassen kan *legge til* nye attributter og metoder.
  - Subklassen kan *overstyre* (override) eksisterende metoder fra superklassen.


Animal
- _name : str
+ speak() : None
Cat
+ speak() : None
Dog
- _breed : str
+ speak() : None

```
1: class Animal:  # Foreldreklasse, arver fra object
2:     def __init__(self, name):
3:         self._name = name
4:
5:     def speak(self):
6:         return "Some generic animal sound"
7:
8: class Dog(Animal):  # Subklasse av Animal
9:     def __init__(self, name, breed):
10:         super().__init__(name)
11:         self._breed = breed
12:
13:     def speak(self): # Overstyrer speak fra Animal
14:         return "Woof!"
15:
16: class Cat(Animal):  # Subklasse av Animal
17:     def speak(self): # Overstyrer speak fra Animal
18:         return "Meow!"
```

Merk `super().__init__()`, for at foreldreklassen skal få name argumentet

Ingen `super().__init__()`: fordi `Cat` ikke har egne attributter som må initialiseres, bruker den arvede `Animal.__init__()`

`super()`

- `super()` metoden brukes når vi har behov for å kalle en metode i foreldreklassen *som heter det samme som en metode i barneklassen*
- Et typisk tilfelle er når vi ønsker å bruke foreldreklassens `__str__()` metode, i stedet for å skrive redundant kode
- På neste slide et eksempel som viser `super()` brukt både i *constructorkall* (`super().__init__(name, age)`) og fra barneklassens `__str__()`

```python
class Person:
    def __init__(self, name, age):
        self._name = name
        self._age = age

    def __str__(self):
        return f"Navn: {self._name}, Alder: {self._age}"

class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self._student_id = student_id

    def __str__(self):
        # Bruker __str__ fra Person og legger til student_id
        return super().__str__() + f", Student-ID: {self._student_id}"

# Eksempelbruk
p = Person("Kari", 45)
s = Student("Ola", 22, "s12345")

print(p)  # Utskrift: Navn: Kari, Alder: 45
print(s)  # Utskrift: Navn: Ola, Alder: 22, Student-ID: s12345
```

Interfaces / Abstrakte klasser (1)

- "Interfaces define *what* should be done, not *how*.", eller "Interfaces are contracts, not implementations."
- Dette uttrykker ideen om at et interface (eller en abstrakt baseklasse i Python) *beskriver hvilke metoder som må finnes, men ikke hvordan de skal implementeres.* Det er opp til hver subklasse å gi sin egen konkrete implementasjon.
- Python har ikke interfaces, slik som en del andre språk, men ved hjelp av mekanismen *abstrakte klasser* kan vi oppnå det samme.

Interfaces / Abstrakte klasser (2)

- *En abstrakt klasse brukes til å definere et felles grensesnitt for en gruppe relaterte klasser*. En abstrakt klasse kan ikke instansieres direkte *og er ment å bli arvet av andre klasser*. Abstrakte klasser kan inneholde både abstrakte metoder (metoder uten implementasjon) og konkrete metoder (metoder med implementasjon).
- *Dersom du arver en abstrakt klasse aksepterer du at du må implementere de metodene som er abstrakte, og du må implementere metodene slik at de gjør det som er formålet*.

Interfaces / Abstrakte klasser (3)

- Hvorfor kan noe slikt være nyttig?
- La oss se litt tilbake på `Animal` klassen:
  - Egentlig er det ingen mening i at klassen `Animal` skal kunne *si noe* (metoden `speak()`).
  - `Animal` er *generisk*, altså et ikke-eksisterende dyr, som kun er der fordi vi ønsker at et `Dog` og et `Cat` objekt skal oppføre seg forskjellig når vi kaller metoden `speak()` på slike objekter.
  - For å få det til, må begge klassene arve `Animal`, samt implementere `speak()`
  - I et slik tilfelle bør vi heller lage en abstrakt klasse, og bruke modulen `abc`

```
1: class Animal:
2:     def __init__(self, name):
3:         self._name = name
4:
5:     def speak(self):
6:         return "Some generic animal sound"
7:
8: class Dog(Animal):
9:     def __init__(self, name, breed):
10:         super().__init__(name)
11:         self._breed = breed
12:
13:     def speak(self):
14:         return "Woof!"
15:
16: class Cat(Animal):
17:     def speak(self):
18:         return "Meow!"
```

`abc` modulen

- `abc`-modulen i Python står for *Abstract Base Classes*, og brukes til å definere abstrakte klasser og metoder – altså klasser som fungerer som *maler* for andre klasser
- **Kort forklart:**
  - ABC er en baseklasse du arver fra for å lage en abstrakt klasse
  - @abstractmethod brukes for å markere metoder som ***må*** implementeres i subklasser
  - Du kan ikke instansiere en klasse som har abstrakte metoder

- **Hvorfor bruke det?**
  - For å tvinge en struktur i subklasser
  - For å sikre at visse metoder alltid finnes i alle underklasser
  - For å lage *interfaces* i Python (selv om språket ikke har egne interface-klasser)

Animal-klassehierarkiet implementert med hjelp av **abc** modulen

```
01: from abc import ABC, abstractmethod
02:
03: class Animal(ABC):
04:     def __init__(self, name):
05:         self.name = name
06:
07:     @abstractmethod
08:     def speak(self):
09:         pass
10:
11:     @abstractmethod
12:     def movement(self):
13:         pass
14:
15: class Dog(Animal):  # Subklasse av Animal
16:     def __init__(self, name, breed):
17:         super().__init__(name)
18:         self.breed = breed
19:
20:     def speak(self):  # implementerer speak
21:         return "Woof!"
22:
23:     def movement(self): # implementerer movement
24:         return "Run"
```

Effekt av å arve `ABC`:
får ikke lov å instansiere `Animal` objekter

Effekt av å bruke dekoratøren `@abstractmethod`:
Barneklasser ***må*** implementere `speak()` og `movement()`

Den abstrakte klassen `Iterator`

- *Design patterns* er et begrep innenfor «computer science» som går ut på at mange problemer egentlig har en felles løsning
- Et velkjent design pattern er «Iterator pattern»
  - Dette design pattern handler om hvordan en kan gjennomløpe en collection / samling data på en uniform måte, uansett hvordan collection'en ser ut
  - Dette design pattern er årsaken til at løkker av typen under fungerer:

```
for elem in [20, 30, 3, 5]:
        <din kode>
```

  - Bak kulissene er et såkalt *Iterator pattern* implementert for `list` klassen, som betyr at `list` klassen har implementert metodene `__iter__()` og `__next__()`
  - `__iter__()` returnerer en *iterator*, mens `__next__()` returnerer *neste element* i collection'en

Iterator interfacet

Python har i modulen `Collections.abc` implementert en abstrakt klasse som implementerer dette design pattern.

Det vi må gjøre er å implementere `__iter__()` og `__next__()` for vår egen collection klasse.

Eksempelet her implementerer `Iterator` interfacet.

Ved å importere den abstrakte klassen `Iterator` fra `collections.abc`, så vil python interpreteren sjekke at begge metodene er implementert i vår klasse `MyIterator` når det instansieres et objekt av den typen.

```
# abstract_base_class_iterator.py
from collections.abc import Iterator

class MyIterator(Iterator):
    def __init__(self):
        self.data = [1, 2, 3]
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index < len(self.data):
            item = self.data[self.index]
            self.index += 1
            return item
        else:
            raise StopIteration

my_iterator = MyIterator()
for item in my_iterator:
    print(item)
```

Utskrift:
1
2
3

Den abstrakte klassen dikterer altså at subklassen ***må*** implementere disse to metodene.

I tilllegg må altså programmereren lese seg opp på hva disse metodene skal gjøre / levere.

Dette er som regel hensikten med en abstrakt klasse, å implementere en avtalt protokoll.

`for` kaller `iter(my_iterator)`, som igjen kaller `my_iterator.__iter__()`

`__iter__()` returnerer `self` (iteratoren selv)

Deretter kaller `for` løkka gjentatte ganger `__next__()` til `StopIteration` blir raised

Hvordan brukes `__iter__()` og `__next__()` fra for løkka?

- Hva skjer når Python utfører en `for`-løkke?

```
for element in iterable:
    <kodeblokk>
```

- Bak kulissene skjer følgende:
  - `iterable.__iter__()` kalles først:
    Python kaller den innebygde funksjonen `iter(iterable)`, som igjen kaller `iterable.__iter__()`. Dette returnerer et *iterator-objekt*
  - Deretter kalles `iterator.__next__()` gjentatte ganger:
    Python kaller `next(iterator)` i en løkke, som igjen kaller `iterator.__next__()` for hvert element, helt til `StopIteration` reises

Alternativ implementasjon (1)
Bruker iteratoren til den aktuelle collection

Her returnerer vi rett og slett iteratoren til den aktuelle collection, her **`list`** klassen sin, og dette er en *ny* iterator som starter fra begynnelsen hver gang.

Dette er den enkleste og mest Pythoniske måten å gjøre en klasse itererbar på, når den allerede inneholder en itererbar struktur

```
class MyCollectionV1:
    def __init__(self, data):
        self._data = data

    def __iter__(self):
        return iter(self._data)  # Bruker lista sin innebygde
  iterator

# Klientkode:
collection = MyCollectionV1([1, 2, 3])
for item in collection:
    print(item, end=" ")

# Kan brukes igjen
for item in collection:
    print(item, end=" ")
```

Utskrift:
```
1 2 3 1 2 3
```

Alternativ implementasjon (2)
Iterator implementert i egen klasse

```
class MyCollectionV2:
    def __init__(self, data):
        self._data = data

    def __iter__(self):
        return MyCollectionIterator(self._data)

class MyCollectionIterator:
    def __init__(self, data):
        self._data = data
        self._index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._index < len(self._data):
            item = self._data[self._index]
            self._index += 1
            return item
        else:
            raise StopIteration
```

```
# Klientkode:
collection = MyCollectionV2([1, 2, 3])
for item in collection:
    print(item, end=" ")

# Kan brukes igjen
for item in collection:
    print(item, end=" ")
```

Utskrift:
1 2 3 1 2 3

Design pattern: Singleton

```
# En singleton klasse sørger for at det kun opprettes ett objekt av klassen
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):  # cls er referansen til klassen
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self, value):
        self.value = value

# Test the Singleton class
s1 = Singleton(10)
s2 = Singleton(20)

print(s1.value) # Output: 10
print(s2.value) # Output: 10
print(s1 is s2) # Output: True (both references point to the same instance
```

`__new__()` blir kalt før `__init__()`, her overrider vi `__new__()` slik at vi hindrer at mer enn ett objekt av `Singleton` typen skapes

Denne kaller `__init__()`

- `cls`: klassen selv (obligatorisk første parameter i `__new__`)
- `*args, **kwargs`: fanger opp alle posisjonelle og navngitte argumenter: gjør `__new__` fleksibel— fungerer uansett hvilke argumenter `__init__` forventer

Python; - arv, polymorfi og duck typing

- Det er faktisk ikke et absolutt krav i Python at polymorfi *må* bruke et klassehierarki eller en felles base class. Python støtter såkalt "duck typing":
  - Hvis to (eller flere) objekter har metoder med samme navn og signatur, kan de brukes om hverandre – uavhengig av arv
- I klassisk OOP (som Java/C++), kreves ofte et felles klassehierarki for polymorfi
- I Python er det vanligst og mest ryddig å bruke en felles base class når du vil ha strukturert polymorfi, spesielt i større systemer
- Uten felles base class kan vi likevel bruke polymorfi så lenge objektene har de nødvendige metodene

Eksempel på polymorfi uten arv:

```
class Dog:
    def speak(self): return "Woof!"
class Cat:
    def speak(self): return "Meow!"

for animal in [Dog(), Cat()]:
    print(animal.speak())
```

Utskrift:

```
Woof
Meow
```

Best practice for polymorf oppførsel er:

- Bruk felles base class (gjerne abstrakt base class med `abc.ABC` og `@abstractmethod`) når du ønsker struktur, feilsjekk og tydelig API for polymorfe objekter
- Bruk duck typing (ingen arv, bare samme metode-navn) for enkle eller små systemer der fleksibilitet er viktigere enn streng struktur
- Dokumenter hvilke metoder som forventes for polymorfe objekter, spesielt hvis du ikke bruker arv
- Foretrekk arv og abstrakte metoder når du vil sikre at alle subklasser implementerer nødvendige metoder

`isInstance` metoden

- `isinstance()`er en innebygd funksjon i Python som brukes for å sjekke om et objekt er en instans av en bestemt klasse (eller en subklasse av denne)
- Dette er nyttig når du har en liste med ulike objekter, og du vil utføre spesielle operasjoner kun på objekter av en bestemt type.

```
class Animal:
    def speak(self):
        return "Some sound"

class Dog(Animal):
    def fetch(self):
        return "Fetching stick!"

class Cat(Animal):
    pass

animals = [Dog(), Cat(), Animal()]

for animal in animals:
    print(animal.speak())
    if isinstance(animal, Dog):
        print(animal.fetch())  # Bare Dog har fetch-metoden
```

Bare Dog har `fetch()`-metoden

Utskrift:

```
Some sound
Fetching stick!
Some sound
```

Multippel arv

Python tillater at en klasse arver fra mer enn én superklasse – dette kalles multippel arv.

Det betyr at en subklasse kan kombinere funksjonalitet fra flere klasser, og dermed få tilgang til attributter og metoder fra alle disse.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound"

class Swimmer:
    def swim(self):
        return f"{self.name} is swimming"

class Flyer:
    def fly(self):
        return f"{self.name} is flying"

class Duck(Animal, Swimmer, Flyer):
    def __init__(self, name):
        Animal.__init__(self, name)

duck = Duck("Donald")
print(duck.speak())
print(duck.swim())
print(duck.fly())
```

Utskrift:
Donald makes a sound
Donald is swimming
Donald is flying

MRO

- Python bruker MRO mekanismen for å avgjøre rekkefølgen metoder og attributter hentes fra når en klasse arver fra flere superklasser. MRO sørger for at Python vet hvilken versjon av en metode som skal brukes, og i hvilken rekkefølge konstruktører og metoder skal kalles.
- Vi kan se MRO for en klasse ved å skrive:

```
print(Duck.__mro__)
```

Utskrift:

```
(<class '__main__.Duck'>, <class '__main__.Animal'>, <class
'__main__.Swimmer'>, <class '__main__.Flyer'>, <class 'object'>)
```

Fallgruber med multippel arv

- Selv om multippel arv kan være nyttig, kan det også føre til kompleksitet og forvirring, spesielt når:
  - Flere superklasser har metoder eller attributter med samme navn.
  - Det er uklart hvilken metode eller attributt som blir brukt – Python bruker **MRO (Method Resolution Order)** for å avgjøre dette.
  - Det oppstår konflikter i konstruktørkall eller attributter.

The diamond problem

- Et klassisk problem ved multippel arv er *the diamond problem*: en subklasse arver foreldreklasse som igjen har en felles foreldreklasse.

- Her arver D fra både B og C, som begge arver fra A. Python løser dette med MRO slik at A.greet() bare kalles én gang.

```
class A:
    def greet(self):
        print("Hello from A")

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass

D().greet()
```

Attributtkonfikt

- Både `A` og `B` klassen har en attributt value. Hvilken vil "overleve"?

- Her vil `self.value` først settes til "A's `value`", og deretter overskrives med "B's `value`". Kan føre til uforutsigbar oppførsel.

Utskrift:
B's value

```
class A:
    def __init__(self):
        self.value = "A's value"
        super().__init__()  # Kaller neste i MRO

class B:
    def __init__(self):
        self.value = "B's value"
        super().__init__()  # Kaller neste i MRO (object)

class C(A, B):
    def __init__(self):
        super().__init__()  # Følger MRO: C -> A -> B -> object

    def show(self):
        print(self.value)

c = C()
c.show()  # Output: B's value
print(f"MRO: {C.__mro__}")  # Viser: C -> A -> B -> object
```

Name mangling

```
class A:
    def __init__(self):
        self.__value = "A's value"
        super().__init__()

    def show_a(self):
        print(self.__value)

class B:
    def __init__(self):
        self.__value = "B's value"
        super().__init__()

    def show_b(self):
        print(self.__value)

class C(A, B):
    def __init__(self):
        super().__init__()  # Følger MRO: C -> A -> B -> object


c = C()
c.show_a()  # Output: A's value
c.show_b()  # Output: B's value
```

Ved å bruke *to understreker* foran attributtnavn kan man unngå kollisjoner.

Python vil endre attributtnavnet for slike fra `__value` til `_A__value` for attributten i `A` klassen, og fra `__value` til `_B__value` i `B` klassen:

Utskrift:

A's value

B's value

Komposisjon som alternativ til multippel arv

```
class A:
    def __init__(self):
        self.a_value = "A's value"

class B:
    def __init__(self):
        self.b_value = "B's value"

class C:
    def __init__(self):
        self.a = A()
        self.b = B()

    def show(self):
        print(self.a.a_value)
        print(self.b.b_value)

C().show()
```

Utskrift:
A's value
B's value

Mix-in klasser kan være et alternativ til multippel arv

```
class LoggerMixin:
    def log(self, message):
        print(f"Log: {message}")

class Processor:
    def process(self):
        print("Processing...")

class LoggedProcessor(Processor, LoggerMixin):
    pass

lp = LoggedProcessor()
lp.process()
lp.log("Done")
```

En mix-in er en liten, spesialisert klasse som gir ekstra funksjonalitet til en annen klasse gjennom arv, uten å være en fullverdig baseklasse.
Mix-ins har vanligvis ingen egen tilstand (ingen `init`), og brukes for å legge til metoder

Utskrift:
Processing...
Log: Done

`Account` utvidet til et klassehierarki med subklasser