Klasser og nedarvning i Python

Kursus 02100 – Indledende Programmering og Softwareteknologi

29. oktober 2025

Jørgen Villadsen og Carsten Witt

Klasser, felter og instansmetoder

I Python bruges nøgleordene `class` (lige som i Java) og `self` (svarende til `this` i Java) til at definere klasser, felter og instansmetoder. Eksempel:

```
class Person:
    def __init__(self, name, cpr_number): # konstruktør
        self.__name = name
        self.__cpr_number = cpr_number

    def get_name(self):
        return(self.__name)

    def get_cpr_number(self):
        return(self.__cpr_number)

    def show(self):
        print("Name:", self.get_name(), "\nCPR number:", self.get_cpr_number())

    def set_name(self, name):
        self.__name = name
    …
```

- Konstruktøren defineres med `__init__`.
- Klassens felter deklareres ikke (man deklarerer ikke variabler i Python), men tilgås med `self.<feltnavn>`.
- Instansmetoder defineres inden for class og skal alle få `self` (den implicitte parameter) som argument.
- Felter, hvis navn begynder med __ (2 underscores), er private (samme princip som i Java) og kan ikke tilgås udefra.

Eksemplet i Java

```
public class Person {

  private String name;
  private String cprNumber;

  public Person(String name, String cprNumber) { // konstruktør
      this.name = name;
      this.cprNumber = cprNumber;
  }

   public String getName() {
      return name;
   }
    …
  public void setName(String name) {
      this.name = name;
   }
  …
}
```

Objekter lavet af klasser

- Skriv `klassenavn(<argumenter til konstruktør>)` ved tildeling, fx opretter `person = Person("Per Sille", "0506012525")`
  et nyt objekt ved navn `person` af klassen `Person`.

- Bemærk, at konstruktøren defineres med en parameter mere (`self`-parameteren er som sagt obligatorisk) end angivet ved oprettelsen af objektet.

- Kald instansmetoder med punktnotation, fx udskriver `person.show()`
  ```
  Name: Per Sille
  CPR number: 0506012525
  ```

- Man kan som sagt ikke direkte tilgå `__`-felterne udefra, fx kaster `print(person.__name)` en fejl.

- Felter, hvis navn ikke starter med underscore, er public og KAN tilgås udefra.

Overlæsning og `__str__`-metoden

- I Python kan man overlæsse (overloade) metoder som i Java.
- Klassisk anvendelsesområde: `toString()`-metode i Java.
- I Python fungerer det på præcis samme måde.
- `print(<objektnavn>)` udskriver som udgangspunkt kryptisk information om objektets lokation i hukommelsen.
- Man skal overlæsse `__str__`-metoden for at få mere sigende information, fx

```
class Person:
    …
    def __str__(self):
        return("Name: " + str(self.get_name()) +  ", CPR: " + \
      str(self.get_cpr_number()))
```

- Nu kan man kalde `print(…)` på objekter af klassen `Person`.

Nedarvning

- I Python virker nedarvning på lignende vis som i Java.
- Skriv navnet på klassen, man ønsker at nedarve fra, i parenteser ved klassedefinition, dvs. `class childclass(<parentclass>)`
- Brug nøgleordet `super` til at tilgå superklassens felter og metoder, herunder konstruktøren.

```
class Student(Person):
    def __init__(self, name, cpr_number, student_id, courses):
        super().__init__(name, cpr_number)
        self.__student_id = student_id
        self.__courses = courses

    def show(self):
        super().show()
        print("Student id:", self.__student_id)
        print("Courses:")
        for i in self.__courses:
            print(i)
```

- Konstruktøren for `Student` kalder konstruktøren for `Person` for at sætte de felter, der er nedarvet.
- Ligeledes kalder `super().show()` instans-metoden `show()` fra superklassen.
- Opret et objekt `student = Student(…)` med 4 argumenter og kald `student.show()`. Udskriver alle 4 felter.
- Feltet `__courses` i dette eksempel er en liste, som for-løkken itererer over.

Eksemplet i Java

```
import java.util.List;
public class Student extends Person {
    private String studentId;
    private List<String> courses;
    public Student(String name, String cprNumber, String studentId,\
 List<String> courses) {
        super(name, cprNumber);
        this.studentId = studentId;
        this.courses = courses;
    }

    @Override
    public void show() {
        super.show();
        System.out.println("Student id: " + studentId);
        …
    }
 …
}
```

Abstrakte klasser

- I stil med *interfaces* i Java findes der såkaldte abstrakte klasser i Python, som man ikke umiddelbart kan lave objekter af. Konkrete klasser skal nedarve fra dem og ”implementere” alle abstrakte metoder fra klasen.

- Tilføj
  `from abc import ABC, abstractmethod` til toppen af filen
  Og brug nøgleordet `pass` i metodens krop samt den såkaldte *decorator* `@abstractmethod`

```
class MyAbstractClass(ABC):
    @abstractmethod
    def my_abstract_method(self):
        pass
```

- For at implementere den abstrakte klasse og den metoder, nedarv fra den på den sædvanlige måde:

```
class MyConcreteClass(MyAbstractClass):
    def my_abstract_method(self):
       print("Implementation of my_abstract_method")
```

Særlige elementer i Python: klasse-variabler

- Java kender statiske variabler/metoder (nøgleord `static`) og instans-variabler/metoder (nøgleordet fraværende). Python kender instans-variabler/metoder og globale (ligner `static`) + en mellemting: klasse-variabler og klasse-metoder.

- Klasse-variabler/metoder er fælles for alle OBJEKTER der er oprettet af klassen.

- Deklarér variabel i klassens virkefelt for at oprette klassevariabel:

```
class Student(Person):
    school = "AB University"
    …
```

- Brug `<klassenavn>.variabel` til at tilgå klassevariabel, fx
  `Student.school = "Technical University of Denmark"`

- Brug af `<objektnavn>.variable` (fx `student.school = ...`, hvor `student` er objekt af klassen `Student`) kan have utilsigtede effekter.

- Anvendelsesområde fx: klassekonstanter, fælles (variable) data for objekter af klasse

Særlige elementer i Python: klassemetoder og destruktører

- Klassemetoder kender ikke til instanser, men kan tilgå klassevariabler:

```
class Student(Person):
    school = "AB University"
    @classmethod
    def get_school(cls):
        return(cls.school)
```

- `get_school` er en klassemetode. Bemærk ”decoratoren” `@classmethod` og brugen af `cls` som reference til klassenavnet. Mangler decoratoren, bliver det til en instansmetode.

- Python kender også destruktører (metodenavn `__del__`), der kaldes, når et objekt er blevet slettet. Fx

```
class Student(Person):
    def __del__(self):
        print("The student has been deleted.")
```

- Man sletter objekter (herunder specifikt klasevariabler, instansvariabler mm.) med nøgleordet `del`, fx

```
student = Student("Anna Gram", ”0102034567", "s23456", []) # opretter objekt
del Student.school # sletter klassevariablen, men beholder resten
del student.publicfield # sletter instansvariablen publicfield
del student # sletter hele objektet og kalder destruktøren
```

Særlige elementer i Python: multipel nedarvning

- I modsætning til Java kan en klasse i Python nedarve fra mere end én superklasse. Angiv bare flere superklasser i argumentet for klassedefinitionen af underklassen.

```
class Student:
    def student_info(self):
        print("I am a student.")

class Lecturer:
    def lecturer_info(self):
        print("I give lectures.")

class TeachingAssistant(Student, Lecturer):
    def ta_info(self):
        self.student_info()
        self.lecturer_info()
```

- Underklassen nedarver alt fra alle superklasser.

- Kald af `ta.ta_info()` på et objekt `ta` af klasse `TeachingAssistant` udskriver bade `I am a student` og `I give lectures`

- Ved navnekonflikter (metoder/felter hedder ens I flere superklasser) bruges den første optræden fra venstre i angivelsen af superklasser. I eksemplet tilgås altså først `Student` ved navnekonflikter.

Dagens opgaver

Python-opgaver om:

- Klasser, herunder instansmetoder, klassevariabler samt konstruktører og destruktører
- Nedarvning
- Abstrakte klasser