DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF COPENHAGEN

Objektorienteret programmering

Datatyper, klasser, objekter og metoder

Kim Steenstrup Pedersen

Plan for denne uge

- Egne datatyper i Python:
  - Abstrakte datatyper
  - Klasser og instanser (objekter)
  - Attributter
  - Funktioner af instanser (med og uden sideeffekter)
  - Metoder og `self`
  - `__init__` og `__str__`
  - Operator overloading
  - Shallow copy versus deep copy
  - Type-based dispatch
  - Polymorfisme på funktionsniveau
  - Klasseattributter
- Objektorienteret programmering I

Klasser og objekter (reminder)

- *Klassen* definerer indholdet og funktionaliteten af datatypen. Indhold kan bestå af indlejret variable som vi kalder *attributter*.
- *Objekter* er konkrete instanser af en klasse og klassen angiver objektets type.
- Når vi opretter et objekt allokeres der plads i hukommelsen til at repræsentere objektets attributter.
- En variabel kan tildeles et objekt og er dermed en *reference* til (et navn for) objektet.
- Flere variable kan pege på det samme objekt. Pas på!

Python benytter call-by-reference til objekter

- Call-by-reference: Ved kald af funktioner og metoder med objekt parametre overføres referencen til objektet – IKKE en kopi af objektet. Pas på!

```
Class Bar(object):
    """En punktklasse"""
    pass
def foo(q):
   """Funktion med sideeffekt"""
   q.x = q.x+10

p = Bar(); p.x=1
foo(p)
print p.x # prints 11
```

Python benytter return-by-reference til objekter

- Return-by-reference: Når funktioner returnere et objekt overføres en reference til objektet – IKKE en kopi af objektet. Pas på!

```
Class MyList(object):
    """En klasse"""
    def getList(self):
      self.L = list()
      return self.L
p = MyList(); q = p.getList()
print p.L, q
q.append(2)
print p.L, q # Nu er både p og q ændret
```

Python benytter call / return–by-value for primitive typer

- Primitive typer, eks. int, float og bool
- Ved parameteroverførsel anvendes call-by-value, dvs. vi får en kopi af værdien ind i funktionen
- Ved returnering anvendes return-by-value, dvs. vi får en kopi af værdien tilbage fra funktionen.

Shallow copy versus deep copy

- Tildelingsoperator = medfører kopiering af reference til instans (kopierer IKKE indholdet af instansen).
- Kopier objekter med `import copy`.
- *Shallow copy*:

```
import copy
p1 = Point()
p2 = copy.copy(p1)
```

- *Deep copy*:

```
import copy
box1 = Rectangle()
box2 = copy.deepcopy(box1)
```

Hvordan tvinger jeg Python til at slette objekter (fjerne dem fra hukommelsen)?

- Der er to metoder:
  `del` eller tildel variablen en anden værdi:

```
class A(object):
    def __del__(self):
        print “Fjerner objekt af typen A: “, self

a=A()
b=A()
b=a
del a
b=A()
b=None
```

Variablers virkefelt (scope) i Python

Pythons variabel virkefelt danner et hierarki:

- Variabler erklæret inden i en funktion er kun tilgængelige inden i funktionen (og indlejrede funktioner).
- Variable erklæret som en attribut på et objekt er kun tilgængelig via objektreferencer (punktum notationen).
- Variable erklæret i et modul er tilgængelig fra alle funktioner og klasser i modulet.
- Variable erklæret i script eller i den interaktive fortolker er erklæret i `__main__` virkefeltet og kan tilgås fra alle funktioner erklæret i `__main__` virkefeltet.
- Og så er der variable erklæret som globale v.h.a. `global` nøgleordet

Globale variable (i global scope)

```
a=1
def foo():
    global a
    a=a+1
foo()
print a # Udskriver værdien 2
```

- Problem: Kan svært at holde styr på hvilke dele af programmet manipulerer `a`.
- Vi behøver ikke globale variable, vi kan I stedet benytte objekter

Plan for denne uge

- Egne datatyper i Python:
  - Abstrakte datatyper
  - Klasser og instanser (objekter)
  - Attributter
  - Funktioner af instanser (med og uden sideeffekter)
  - Metoder og `self`
  - `__init__` og `__str__`
  - Operator overloading
  - Shallow copy versus deep copy
  - Type-based dispatch
  - Polymorfisme på funktionsniveau
  - Klasseattributter
- Objektorienteret programmering I

Type-based dispatch eller type-check-i-hånden

- Vi kan benytte funktionen `isinstance(object,type)` til at undersøge om `object` har typen `type`

- Addition mellem kompleks tal og reelt tal kan se således ud:

```
def __add__(self,other):
        """Lav en ny kompleks variable, som er summen af
            self og other"""
        if isinstance(other, kompleks):
          return kompleks(self.re()+other.re(),self.im()+other.im())
        else:
          return self + kompleks(other, 0.0)
```

- Dette kaldes type-based dispatch.

Polymorfisme på funktionsniveau

- Da vi har implementeret operatorene kan vi anvende nogle standard funktioner med vores kompleks klasse.
- Eksempel:

```
L=[Kompleks(1,0), Kompleks(2,1), Kompleks(3,2)]
print sum(L)
```

6 +i3.0

- Funktioner som kan arbejde på objekter af forskellige typer kaldes *polymorfe*.
- Det kan lade sig gøre fordi kompleks klassen opfylder den *grænseflade* `sum()` forventer, dvs. at additionsoperatoren er defineret.

Klasseattributter

- Vi kan definere attributter som er tilknyttet klasse objektet:

```
class A(object):
      mystr = "min klassevariabel"
```

- Kan tilgås uden om objekt instanser:

```
print A.mystr
a = A()
print a.mystr
a.mystr="En ny streng"
print A.mystr
print a.mystr
```

Adgangskontrol
ADVARSEL: Avanceret manipulation af objekter

- Ok, vi kan faktisk forhindre dynamisk manipulation af attributter:

```
class A(object):
    """This class disallows dynamic alteration of its attributes""”
    def __init__(self, a):
        self.a = a
    def __setattr__(self, name, value):
        if name == 'a': # Ellers så virker __init__ ikke
            object.__setattr__(self, name, value)
        else:
            raise AttributeError('Unknown attribute’)
    def __delattr__(self, name):
        raise AttributeError('Cannot delete attributes’)
a = A(1)
a.a = 10
print a.a
a.b = 10 # AttributeError: Unknown attribute
del a.a # AttributeError: Cannot delete attributes
```

Objektorienteret programmering

Nøglekoncepter:

- Klasser
- Instanser
- Metoder
- Polymorfisme
- Abstraktion
- Indkapsling
- Grænseflade specifikation
- Arv
- Dynamic dispatch

Objektorienteret programmering

Nøglekoncepter:

- Klasser
- Instanser
- Metoder
- Polymorfisme
- Abstraktion
- Indkapsling
- Grænseflade specifikation
- Arv
- Dynamic dispatch

Objektorienteret programdesign

Husk fra Knuds forelæsning om afprøvning og struktureret programmering

Objektorienteret problemløsning

- Bliv grundig bekendt med problemets domæne. Hvilke begrebsdannelser, relationer, lovmæssigheder er der?
- Vælg en konsekvent og systematisk terminologi.
- Situationen har analogier til udforskning af et nyt videnskabeligt felt.
- Ved objektorienteret analyse afgrænser man
  1. De *objekter*, som skal håndteres (punkter, forsøgsresultater, biologiske arter, samfund, . . .)
  2. De *metoder*, som hører til disse objekter. I Python skal alle objekter have en initialiseringsmetode og en udskrivningsmetode, men derudover står mulighederne åbne (flytning, sammenligning, fletning, udtræk, . . .)

Abstraktion

- Repræsentation af centrale koncepter fra problem domænet.

- Eksempel:
  Vores kompleks klasse repræsenterer konceptet komplekse tal og laver en abstraktion som kan anvendes i andre abstraktioner (eks. komplekse bølgefunktioner).

- Eksempel:
  Konceptet observationer fra et eksperiment kan repræsenteres med en `Observation` klasse. Konkret indsamlet data repræsenteres som en instans af klassen `Observation`.

Indkapsling

- Skjul irrelevante detaljer om en abstraktion (klasse) fra brugeren af denne.
- Brugeren af din klasse behøver ikke at kende din datarepræsentation.
- Definer i stedet en grænseflade der tillader brugeren at manipulere data uden at kende repræsentationen.
- Fordel: Du kan udskifte datarepræsentation og du minimere fejl under data manipulation.

Indkapsling i Python

- I Python er alle metoder og attributter (både instanser og klasse attributter) tilgængelig for alle. Andre sprog indeholder mekanismer til adgangsbegrænsning: Eks. public, private og protected i C++ og Java.
- Undlad at tilgå attributter direkte, benyt i stedet metoder til at tilgå disse. Sådan sikre du at du kan udskifte repræsentationen i klassen.
- Undlad at tilføje attributter til objekt udenfor klassedefinitionen – kan medfører at klassens metoder begynder at fejle.

Grænseflade specifikation

- Grænseflade specifikation hentyder til at klassens metoder og attributter definerer en grænseflade (interface) for hvordan vi skal tilgå objekter.

- Et objektorienteret program kan betragtes som en samling af interagerende objekter i modsætning til en række sub-rutiner / funktioner som skal udføres sekventielt.

- Objekter interagere ved at kalde metoder på hinanden – kaldes også at sende beskeder (message passing).

Objektorienteret programmering (og design)

Nøglekoncepter:

- Klasser
- Instanser
- Metoder
- Polymorfisme
- Abstraktion
- Indkapsling
- Grænseflade specifikation
- Arv
- Dynamic dispatch

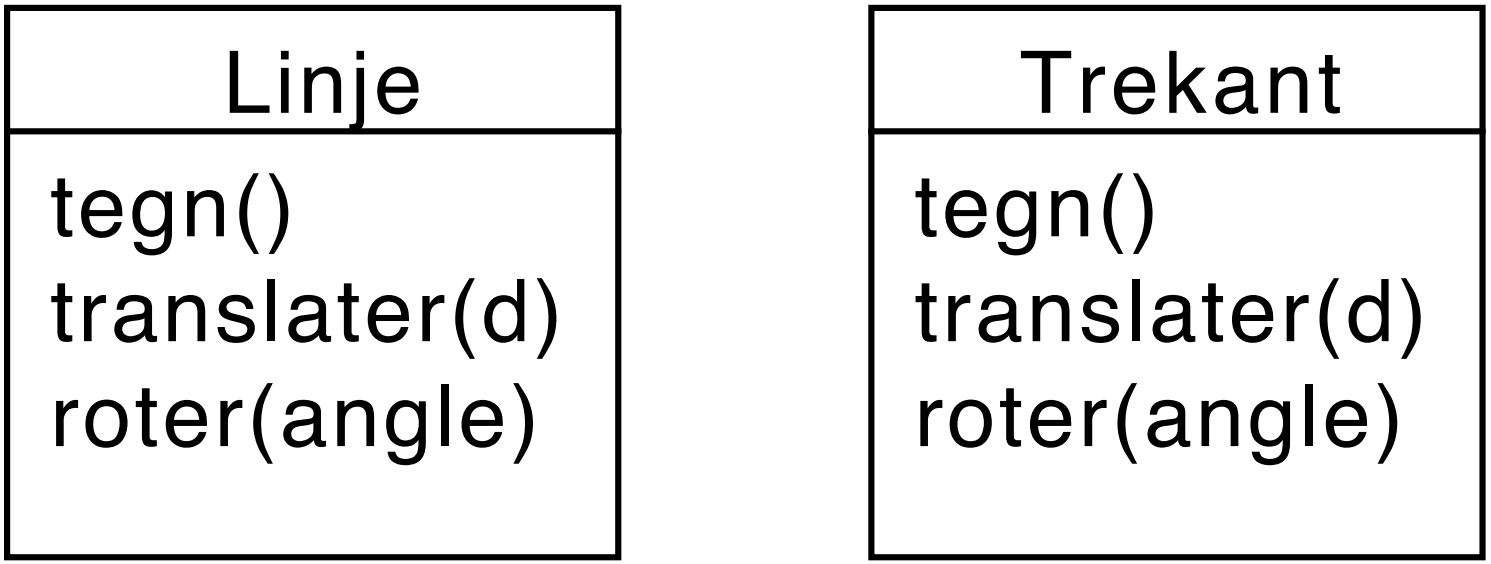
Objektorienteret analyse og design: Eksempel fra computergrafik

- Repræsentation og manipulation af geometriske objekter såsom punkter, linjer og trekanter.
  Vi skal som minimum kunne
  - Transformere koordinaterne (translation, rotation, skalering)
  - Tegne det geometriske objekt

- Hvilke klasser har vi behov for og hvordan skal deres grænseflade se ud?

- Lad os kigge på `form.py`.

Objektorienteret analyse og design: Eksempel fra computergrafik



Resumé

- Python understøtte det objektorienteret programmerings paradigme

- Definition af egne typer:
  - Klasser og instanser
  - Attributer og metoder
  - Operator overloading
  - Og andre detaljer

- Nøglekoncepter for objektorienteret programmering

A word from our sponsors …

Måling af trivsel og tilfredshed

KU undersøger studiemiljøet på din uddannelse.

Fortæl os, hvordan vi kan forbedre din hverdag som studerende!

Husk at svare på undersøgelsen senest 3. december.

Du får link til undersøgelsen via din KU-mail, og du kan læse mere om undersøgelsen på KUnet.

Tak for dit svar ☺