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

Egenudviklet (komplekse) datatyper

- Vi kan konstruerer nye komplekse datatyper ud fra de indbyggede datatyper.
- I Python skal vi benytte *klasser* og *objekter* til dette.
- Komplekse datatyper kan inddeles i:
  - Problemspecifikke datastrukturer og tilhørende funktionalitet
  - Abstrakte datatyper som er generelt anvendelige (ikke problemspecifikt)
- Abstrakte datatyper:
  - Definition af generelt anvendelige datatyper med et veldefineret sæt af tilladte operationer (uafhængig af programmeringssprog).

Abstrakte datatyper: Eksempler fra Python

- Lister:
  - Operationer: Opret liste, indsæt element, læs element, længde af listen, …

```
L = list()
L.append(1); L.append(2)
L.insert(1,3)
print L[1], len(L)
```

- Hash-tabeller (Dictionaries):
  - Operationer: Opret tabel, indsæt element, læs element, …

```
D = {'spam': 1, 'eggs': 2}
D['ham']=3
print D['spam']
```

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, . . .)

Objektorienteret programmering

Nøglekoncepter:

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

Egne datatyper i Python

- Klasser
- Instanser (objekter)
- Attributter
- Objekter kan ændres (mutable)
- Funktioner af instanser (med og uden sideeffekter)
- Metoder og `self`
- `__init__` og `__str__`
- Operator overloading

Klasser og objekter

- *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!

Definition af en klasse

- I Python 2.X er der forskel på følgende klasse definition:

```
class A():
    pass

class B(object):
    pass
```

- Klasse `B` er *new-style* klasse definition som medfører at klasser og instanser håndteres på linje med indbyggede typer. Klasse `A` er *classic-style* klasse definition.
- Godt råd: Benyt altid new-style klasse definitioner, dvs. husk (object) efter klassenavnet.

Egne datatyper

- Klasser
- Instanser (objekter)
- Attributter
- Objekter er mutable (kan ændres)
- Funktioner af instanser (med og uden sideeffekter)
- Metoder og `self` (en reference til objektet selv)
- `__init__` og `__str__`
- Operator overloading

Specielle metoder

Python har en række specielle metoder som kan benyttes til at modificere Pythons håndtering af dine egne klasser:

- `__init__(self [, arg]* )`:
  Konstruktør metode der kaldes ved oprettelse af instans af klassen (objektinstansering).
- `__str__(self)`:
  Kaldes ved `str(self)` eller `print self` og skal returnere en strengrepræsentation af objektet.
- `__del__(self)`:
  Destruktør metode som kaldes når et objekt ikke længere refereres til. Benyttes til oprydning og deallokering af ressourcer (eks. luk filer)

Eksempler på operatorer

- Implementer følgende metoder for din klasse for at benytte operatorer på din klasse:

```
__add__(self, other)      self + other
__sub__(self, other)      self - other
__mul__(self, other)      self * other
__div__(self, other)      self / other
__pow__(self, other)      self** other
```

- Find flere operatorer og specielle metoder i Python Language Reference.

Sammenligningsoperatorer

- Variant 1: Definer følgende operator metoder så de returnere en boolsk værdi

```
__lt__(self, other)      self <  other
__le__(self, other)      self <= other
__eq__(self, other)      self == other
__ne__(self, other)      self != other
__gt__(self, other)      self >  other
__ge__(self, other)      self >= other
```

- Variant 2 (kun Python 2.x): Definer operator metoden `__cmp__(self,other)` således at den returnere -1, 0 eller 1 hvis self er < other, self == other eller self >other.

Sammenligningsoperatorer og objekter

- Operator `is` er sand hvis objekt instanser er identiske.
- Operator == for selv-defineret klasser er identisk med `is` operator medmindre `__eq__` eller `__cmp__` er defineret:

```
p1 = Point(1, 2)
p2 = p1
p1 is p2
True
p2 = Point(1, 2)
p1 == p2
True    # Hvis __eq__ er defineret i Point
p1 is p2
False
```

Abstrakt datatype eksempel: Komplekse tal

- Lad os implementere vores egen repræsentation af komplekse tal.

- Regneregler: z1 = a + b i og z2= c + d i
  - Addition: z1 + z2 = (a + c) + (b + d) i
  - Subtraktion: z1 - z2 = (a - c) + (b - d) i
  - Kompleks konjugation: z1^* = a – b i
  - Multiplikation: z1 * z2 = (a * c - b * d) + (b * c + a * d) i
  - Division: z1 / z2 = (a * c + b * d) / (c^2 + d^2) + (b * c - a * d) / (c^2 + d^2) i

- Hvordan skal programmet se ud?
  (Jeg har snydt og forberedt test kode)

Hvad med addition mellem komplekse tal og et reel tal?

- 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.

Men det virker ikke helt endnu

- z1+ 2 OK Kalder `z1.__add__(2)`
- 2 + z1 FEJL Kalder `z1.__radd__(2)`

- Løsningen er at implementere `__radd__` metoden:

```
def __radd__(self,other):
      """Lav en ny kompleks variable, som er summen af
          self og other"""
      return kompleks(other) + self
```

Mandlebrots mængde

Benoît Mandelbrot (1924-2010), Foto: Rama, wikipedia

- Det komplekse tal c er en del af mandelbrots mængde, hvis

$$\forall j : \left| z_j \right| < konst.$$

- når

$$z_j, c \in C$$

$$z_0 = 0$$

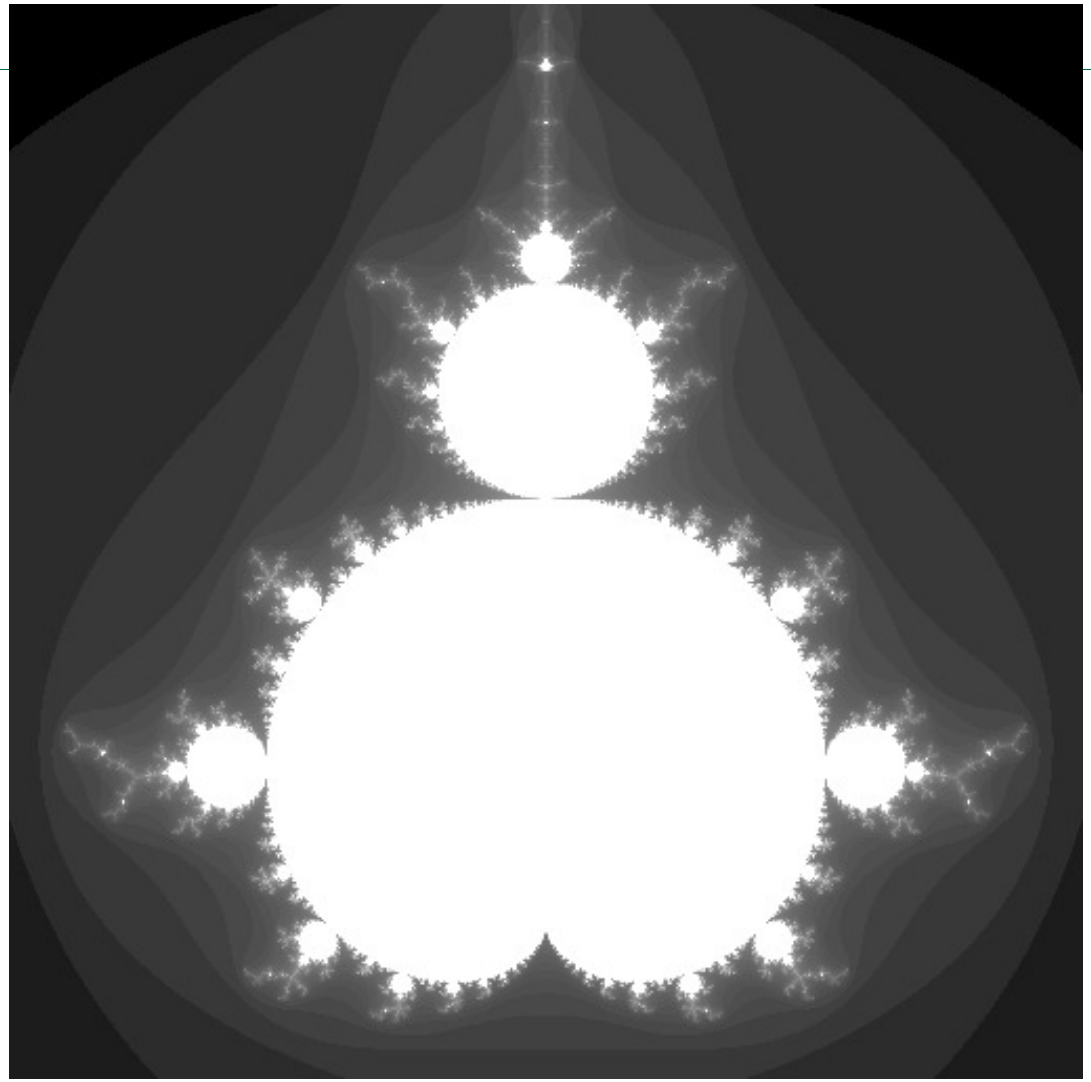
$$z_{j+1} = z_j^2 + c$$

- Alternativt, c tilhører ikke Mandelbrots mængde, hvis

$$\left| z_j \right| > 2$$

Mandlebrots mængde i Python

```
import math, sys

# Interval af c-konstanten, som der undersoges:
CxMin = -1.9
CxMax = 0.6
CyMin = -1.25
CyMax = 1.25

# Resultatbilledets storrelse (NxN)
N = 512;

# Maksimal antal iteration per pixel
iterMax = 512

# Fil til resultatbilledet
fp = open("mandelbrot.pgm", 'w')

# Udskriv header information for pgm billedstandarden
fp.write("P2 %d %d %d\n"%(N,N,255))

for m in range(N):
    # Skaler m i 0..N-1 til Cx i CxMin..CxMax
    Cx = CxMin + (CxMax-CxMin)*m/(N-1)
    for n in range(N):
        # Skaler n i 0..N-1 til Cy i CyMin..CyMax
        Cy = CyMin + (CyMax-CyMin)*n/(N-1)
        I = iterMax
        Zx = 0
        Zy = 0
        for iter in range(iterMax):
            # Z^2 = (Zx+i*Zy)*(Zx+i*Zy) = (Zx^2-Zy^2)+i*(2*Zx*Zy)
            Tx = Zx*Zx-Zy*Zy + Cx
            Ty = 2*Zx*Zy + Cy
            Zx = Tx
            Zy = Ty
            # Saasnart laengden af det komplexe tal overstiger 2 er
            # divergensen sikker.
            if Zx*Zx+Zy*Zy >= 4.0:
                I = iter
                break
        # Udskriv pixel vaerdien, paa en logaritmisk skala
        fp.write(" %d"%(255*math.log(I+1)/math.log(iterMax+1)))
    print "%d\r"%m,
    sys.stdout.flush()
    fp.write("\n")
fp.close()
```



God programmeringsskik i Python

- Første parameter i metode definitionen er en reference til objektet som metoden anvendes på. Behøver ikke at hedde `self`, men det er en god ide at benytte denne standard.
- Benyt konstruktørmetoden til at allokerer og initialiserer attributter, samt at initialiserer ressourcer som eksempelvis at åbne filer.
- Benyt destruktørmetoden til at ryde op i attributter, eksempelvis til at lukke åbne filer.
- Benyt strengmetoden til at hjælpe med fejlfinding i programmer og til at kunne udprinte læsevenlige overblik af datatypen.

God programmeringsskik i Python

- Vi kan dynamisk tilføje attributter og metoder til et objekt efter instantiering. Det er generelt ikke en god ide!
- Alle attributter og metoder er i Python tilgængelige og kan overskrives. Stå i mod fristelsen!
- Dynamisk tilføjelse og ændring af attributter og metoder kan ødelægge en implementation (få programmet til at fejle), specielt hvis du gør det på kode du ikke selv har skrevet.