



Föreläsning 7  
 Programmeringsteknik och C DD1316

- Egen modul
- Klassattribut
- Klassmetoder
- Metoderna: `__str__`, `__lt__`, ...
- Arv
- Överlagring av metoder



Egen modul

- Man kan samla alla funktioner och klasser i en fil och importera filen i huvudprogrammet, precis på samma sätt som man importerar moduler från standardbiblioteket.

Ex:

geometry.py:
 

```
class Circle (object):
    def __init__ (self, r):
        self.radius = r
        ...
```

myprogram.py:
 

```
import geometry
c1 = geometry.Circle (3)
```

2/12



Klassattribut

- Ett attribut som tillhör klassen och inte objektet.

```
class Bag (object):
    nBags = 0 # antal påsar vi skapat

    def __init__ (self, nApples):
        self.nApples = nApples
        Bag.nBags += 1 # ny påse

    def apples (self):
        return self.nApples
```

- Access från klassobjektet:  
`Bag.nBags`
- Åtkomlig från instanser (men tillhör klassobjektet):  
`bag = Bag ()`  
`bag.nBags`

3/12



Klassmetod

- En metod som tillhör klassen och inte objektet.

```
class Bag (object):
    _nBags = 0 # antal påsar vi skapat

    def __init__ (self, nApples):
        self.nApples = nApples
        Bag._nBags += 1 # ny påse

    def apples (self):
        return self.nApples

    @staticmethod
    def nBags ():
        return Bag._nBags
```

4/12



Speciella metodnamn

- Vissa metodnamn i klasser har en speciell betydelse (se "special method names" i Python-dokumentationen)
- `__init__` (self, ...) är konstruktorn som anropas då en instans skapas
- Hur kan vi välja en egen strängrepresentation av ett objekt?

```
>>> Complex (3, 4)
<__main__.Complex object at 0x7f6d07f9ceb8>

Vill ha:

3+4i
```

- För detta kan vi använda `__str__` (self)



`__str__`

- `__str__` (self)  
används om man vill definiera hur en strängrepresentation av objektet ser ut (%g i formateringen ger enklast möjliga representation av talet, heltal eller flyttal)

```
class Complex (object):
    def __init__ (self, re, im):
        self.re = re
        self.im = im

    def __str__ (self):
        return "%g+%gi" % (self.re, self.im)
```



## Aritmetiska operationer

- Vi vill kunna addera två komplexa tal  $a$  och  $b$  med  $a + b$
- ```
class Complex (object):
    ...
    def __add__ (self, other):
        return Complex (self.re + other.re,
                        self.im + other.im)
```
- $a + b \rightarrow$  `__add__ (a, b)`
  - `__add__` ska därför returnera `self + other`
  - För `-`, `*` och `/` finns `__sub__`, `__mul__`, `__truediv__`



## Speciella metodnamn för ordningsrelationer

- `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__` och `__ne__` används om man vill jämföra två objekt (oftast av samma typ) med, i samma ordning, följande jämförelseoperatorer: `<`, `>`, `<=`, `>=`, `=` och `!=`
- `__lt__` används av `sort`
- Exempel (ordning enligt magnitud):

```
class Complex (object):
    ...
    def __lt__ (self, other):
        return magnitude (self) < magnitude (other)
```

8/12



## Arv

- I objektorienterade språk finns ofta en arvsmekanism
- *Subklass* ärver från *superklass*
- En subklass ärver alla metoder och attribut från superklassen
- Superklassens konstruktor kan anropas från subklassens konstruktor via:

```
super (<subklass>, self).__init__ (...)
```

9/12



## Exempel

```
import math

class Ellips (object):
    def __init__ (self, l, k):
        self.lang = l
        self.kort = k

    def area (self):
        return math.pi * self.lang * self.kort

class Cirkel (Ellips):
    def __init__ (self, r):
        # anrop av Rektangels konstruktor:
        super (Cirkel, self).__init__ (r, r)
```

10/12



## Överlagring av metoder

- *Överlagring* av en metod är då man definierar en metod i en subklass och en metod med samma namn redan är definierad i superklassen

11/12



## Exempel

```
class Parallelogram (object):
    def __init__ (self, a, b, h):
        self.kant = a
        self.basKant = b
        self.hojd = h

    def area (self):
        return self.hojd * self.basKant

    def omkrets (self):
        return 2 * (self.kant + self.basKant)
```

12/12



## Exempel

```
class Rektangel (Parallelogram):
    def __init__ (self, a, b):
        self.kant = a
        self.basKant = b

    def area (self): # Överlagring av area i super
        return self.kant * self.basKant

class Kvadrat (Rektangel):
    def __init__ (self, a):
        super (Kvadrat, self).__init__ (a, a)
```

13/12



## Utökning

- Arv är användbart när man vill skapa en mer specialiserad version av superklassen.
- Man *utökar* en subclass om man i klassen lägger till metoder eller attribut som inte är deklarerade i superklassen.

14/12



## Sammanfattning

- Dela upp ditt program i olika filer och använd moduler för att införa funktioner och klasser
- Klassobjekt kan ha klassattribut och klassmetoder
- `__str__` är en metod som anropas när man använder objektet i `str ()` och `print ()`
- `__lt__` är en metod som anropas när man använder objekt i jämförelseoperatorer, användbar i samband med funktionen `sort ()`
- Arv används för att skapa specialiserade subclasser utifrån mer allmängiltiga superklasser
- Subklassen ärver attribut och metoder från superklassen

15/12