Datastrukturer, procedurell- och dataabstraktion

Human Centered Systems
Inst. för datavetenskap
Linköpings universitet

Översikt – 5 delar

- Fysisk talrepresentation och teckenkodning
- Datastrukturer och strukturdelning
- Modellering och abstraktion
- Abstrakta datatyper (ADT)
- Procedurell abstraktion och högre ordningens funktioner

Relaterade avsnitt: LP 5,6,15, 17 PL 11.1-11.3

1: Fysisk representation

Binär, oktal och hexadecimal representation

- Datorn representerar "allt" som binära tal, dvs sekvenser av 0 och 1
  - det binära positionssystemet: 2^0, 2^1, 2^2,...
- Jämför decimala positionssystemet:
  - Decimalt: 234 = $2*10^2 + 3*10^1 + 4*10^0$
  - Basen är 10 $\quad 7 \quad 6 \quad 5 \quad 4 \quad 3$
  - Varje position har en viss vikt 0, 1, 2 ... (från vänster)
- Programmering: normalt Hex och oktalt (istället för binärt)

234 uttryckt i andra system

Decimalt: 234 = $1*2^7 + 1*2^6 + 1*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0$

Binärt 11101010

Binärt 011 101 010
Oktalt: 3 5 2
Decimalt 234 = $3 * 8^2 + 5 * 8^1 + 2 * 8^0$

Binärt 1110 1010
Hex: E A
Decimalt 234 = $14 * 16^1 + 10 * 16^0$

Oktalt och hexadecimalt i Python

```
>>> 0352
234
>>> 0xEA
234
>>> 0xEA + 0352
468
>>> 0359
  File "<stdin>", line 1
 0359
    ^
SyntaxError: invalid token
>>> 0xEA.45
  File "<stdin>", line 1
    0xEA.45
        ^
SyntaxError: invalid syntax
>>> 0xEA & 0xF
10
>>> '%X' % 234
'EA'
```

Tal är tal...oavsett kodning

9 inte oktal

inga decimaler

OBS: skilj på hur datorn representerar fysiskt och Pythons **literaler**... här visas literaler – inte **fysisk representation**

www.unicode.org

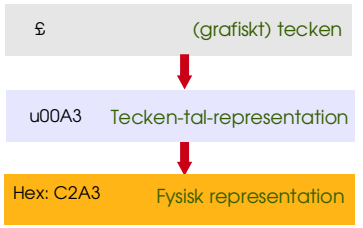
- Unicode en global teckenmängd
  - men igen datorrepresentation specifierat
  - U+xyzu där xyzu är hexa-decimalt tal (16-bit)
- Flera kända implementationer finns
  - Unicode Transformation Format: UTF-8, UTF-16, ...
  - Universal Character Set: UCS-2, UCS-4, ...
- Unicode-nivån är bättre än konkret teckenkodning
  - Exportera till specifik kodning vid I/O istället

Bättre

```
>>> print u'\u00A3'
£
>>> print '\xc2\xa3'
£
```

Sämre

Mer om teckenkodning



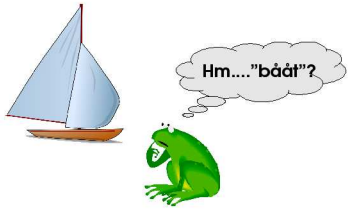
Exempel: Teckenkodning i Python

```
>>> print u'\u00A3'
£
>>> print '\xc2\xa3'
£
```

Anger unicode-tecknet
Föredra när det går!

Anger UTF-8-kodningen direkt
Ger plattformsberoende –
rätt tecken endast på UTF-8-datorer

2: Modellering och abstraktion



Modellering

Urval av aspekter

Den verkliga situationen ⟺ **Programmets modell av situationen**

first_name = Fia
last_name = Ohlsson
age = 34

Abstraktion

- Tankemässig generalisering från objekt
- Information om är mer eller mindre konkret/abstrakt
  - bilen är **röd**
  - fåglar har **vingar**
  - cyklar har **två hjul**
  - Emil är **man**; Maria E en **kvinna**

Organism
Varelse
Djur
Fågel

Viktiga grundbegrepp

- Program löser en uppgift
- Situationen för uppgiften måste modelleras
- Programmering med bra abstraktion gör lösningen enkel och begriplig

*Tänk om jag skulle göra ett progam utan specifik uppgift, som inte modellerar något och som saknar abstraktion...**det vore väl kul!?***

3: Datastrukturer och strukturdelning

Begreppet objekt/datastruktur

- LP-boken använder objekt synonymt med data
  - Objektstrukturer kanske bättre...men mindre standard
- Ett programs datastruktur är den representation vi väljer när vi modellerar verkligheten
- Våra byggstenar är objekt/datatyperna i programmeringsspråket
- Datastrukturen väljs utifrån vad som är praktiskt för programmet/programmeraren
- Enkelhet och effektivitet nyckelord

Hur bestämma datastrukturer?

- Studera tillämpningen
- Identifiera lämpliga datatyper
- Välj ut vilka aspekter i tillämpningen som ska modelleras
- Sammansatta strukturer: välj ut de ingående delarna och hur man bäst sätter samman dem
- Identifiera vilka operationer som ska stödjas av strukturen (jmf. ADTer strax)

Strukturerad modellering

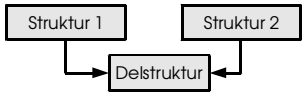
document.py:

```
load_file(path)
find_lines(word, document)
```

```
>>> doc = load_file('my-doc.txt')
>>>print doc
[['Detta', '\xc3\xa4r', 'ett',
'test.'],...['Test', 'igen!']]
>>> lines = find_lines("test",
doc)
>>> print lines
[0, 7]
```

- Hur representera en textfil internt i ett program?
- Förslag: **lista av listor av strängar**
- Strukturen stödjer vissa funktioner bra men inte alla
- Begriplig struktur
- Lite ineffektivt att t ex söka på ord/strängar...men fungerar

Strukturdelning



- Används när **samma data** finns i **flera roller**
- Användbart när samma information ska finnas i flera kontext
  - T ex samma "person" ingår i flera "projekt"
- Strukturdelning undviks med kopiering
  - Viktigt att vi inte strukturdelar av misstag!
- Vanlig avancerad form av datastrukturering

Exempel på strukturdelning

```
>>> person1 = {"name": "Fia", "age" : 34}
>>> person2 = {"name": "Olle", "age" : 36}
>>> person3 = {"name": "Stina", "age" : 22}
>>> members1 = [person1, person3]
>>> members2 = [person1, person2]
>>> person1["age"] = 35
>>> members1
[{'age': 35, 'name': 'Fia'},
 {'age': 22, 'name': 'Stina'}]
>>> members2
[{'age': 35, 'name': 'Fia'},
 {'age': 36, 'name': 'Olle'}]
```

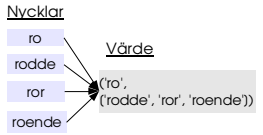
***Exemplet utläst**: vi har två listor som representerar medlemmar (i t ex projekt). Samma personer förekommer i flera av listorna. Genom att dela struktur kan vi ändra personernas data globalt enkelt.*

Källkodsexempel: lexicon.py

```
lexicon = new_lexicon()
add_term('ro', lexicon)
add_form('rodde', 'ro', lexicon)
add_form('ror', 'ro', lexicon)
add_form('roende', 'ro', lexicon)
print get_term('ror', lexicon)
print get_term('rodde', lexicon)
```

```
('ro', ['rodde', 'ror', 'roende'])
('ro', ['rodde', 'ror', 'roende'])
```

dictionary-struktur:



4: Abstrakta datatyper (ADT)

Begreppet Abstrakt datatyp (ADT)

- Egna införda funktioner för en viss typ av data – kallas ADTn primitiva operationer/primitiver
- I Python lämpligen i en egen modul
- Arbetssätt:
  - Bestäm datastruktur
  - Bestäm primitiver
  - Implementera primitiverna
- Styrkan med en ADT är att internt dataformat kan ändras utan att påverka anropande källkod
- Anropande källkod kan inte heller ändra dataformatet vilket gör ger en ökad säkerhet

Vanliga primitiver i en ADT

- **Konstruktorer**: skapar ett nytt ADT-objekt
- **Selektorer**: väljer ut delar ur ADTn
- **Igenkännare**: testar om ett okänt objekt tillhör en viss ADT
- **Iteratorer**: går igenom alla delelement i en ADT
- **Modifikatorer**: förändrar datat som finns i en viss ADT

Exempel på ADT: person

person.py

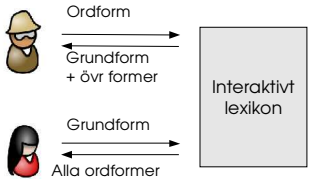
```
def create(name, age)
def set_name(p, name)
def get_name(p)
def set_age(p, name)
def get_age(p)
def has_same_name(p1, p2)
def has_same_age(p1, p2)
```

Testkod:

```
p1 = create("Micke", 44)
p2 = create("Bosse", 44)
p3 = create("Karin", 30)
p4 = create("Micke", 30)
print has_same_name(p1, p2)
print has_same_age(p1, p2)
print has_same_name(p1, p4)
```

```
False
True
True
```

Exempel-2: lexikon.py



- Lexikon: uppslagning baserat på förekomst mot grundform
- Vi vill att man ska kunna slå upp på valfri form
- Lexikonet ska ha **en informationspost** per ord

Exempel på förfrågningar:
"Vilka ordformer finns för ordet '**ror**'?"
"Vilka ordformer finns för ordet '**rodde**'?"

ADT lexikon.py

lexicon.py:

```
new_lexicon()
add_term(concept, lexicon)
add_form(form, concept, lexicon)
get_term(form, lexicon)
get_all_forms(lexicon)
```

```
lexicon = new_lexicon()
add_term('ro', lexicon)
add_form('rodde', 'ro', lexicon)
add_form('ror', 'ro', lexicon)
add_form('roende', 'ro', lexicon)
print get_term('ror', lexicon)
print get_term('rodde', lexicon)
```

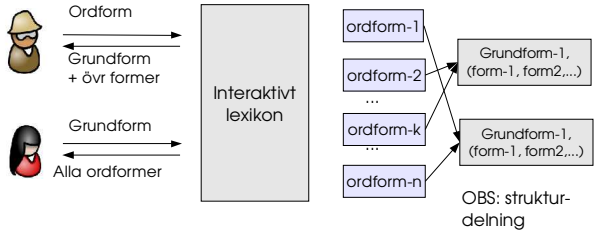
- ADT med operationer för att
  - skapa ett lexikon
  - Lägga till ord/böjningar
  - Slå upp ord/böjningar
- Varje begrepp kallar vi en "term" i lexikonet
- Grundordet för concept
- Böjningar för "forms"

Datastrukturen beror på implementationen, t ex:

```
('ro', ['rodde', 'ror', 'roende'])
('ro', ['rodde', 'ror', 'roende'])
```


Datastruktur för ADTn lexikon.py

Intern datastruktur:
en dictionary där alla böjnings-former är nycklar.



OBS: struktur-delning

5: Procedurell abstraktion och högre ordningens funktioner

Hm har jag gjort funktioner av lägre ordning tro...!?

Procedurell abstraktion med funktioner

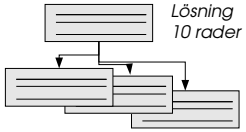
- Ytterligare en (mer komplex) kontrollstruktur
- Namngivet källkodsblock
  - deklareras ungefär som en variabel
  - vi anropar subprogrammet (eng invoke)
  - vi kan skicka med parametrar
  - kontrollen returneras när subprogrammet kört klart
- Kallas även *subrutiner* (eng subroutines)
- De flesta programmeringspråk har någon form av subprogram-konstruktion

Varför abstrahera med funktioner?

Allt som en funktion

Lösning
50 rader

Uppdelning i flera funktioner



Lösning
10 rader

- Uppdelning av problem i delar
- Det totala problemet löses på en abstraktare nivå
  - genom anrop till underprogram
- Införande av funktioner innebär abstraktion av lösningens *beskrivning*
- *Generalitet: samma kod kan återanvändas*

Ökad läsbarhet...har vi redan provat på

Exempel på huvudprogram:

```
Input = read_input()
results = calculate_results()
present_results(results)
```

- Huvudprogrammet liknar pseudokoden
- Enkelt för en oinsatt att hitta in i källkoden
- Ändringar kan ske utan att förstå hela programmet
- Delar upp programmet i funktionella delproblem

Polymorfism – ett sätt att öka återanvändningen

```
def intersect(list_a, list_b):
    res = []
    for el in list_a:
        if el in list_b:
            res.append(el)
    return res
```

- Generella funktioner ger återanvändbar kod
- Delar upp programmet i abstrakta och konkreta delar

```
intersect([1,2,3,4],[0,2,4,6]) ⟹ [2, 4]
intersect([1,2,3,4],(0,2,4,6)) ⟹ [2, 4]
intersect(['a','b','c','d'],'Acdfgh') ⟹ ['c', 'd']
```

Funktioner är objekt

```
>>> def foo(): pass
>>> foo
<function foo at 0x83370d4>
```

- Funktioner är objekt liksom alla andra objekt
  - vi kan skicka runt dem i ett program
  - de har en identitet
  - olika namn kan ha samma identitet/funktionsobjekt
- Funktioner är speciella för att de kan evalueras m a p attribut - apply/call-funktionen
- Genom att skicka runt funktioner ökar vi den procedurella abstraktionen

Exempel: hantering av funktionsobjekt

```
>>> def add_one(x): return x + 1
...
>>> fn = add_one
>>> fn(3)
4
>>> apply(add_one, [9])
10
>>> args = [9]
>>> fn(*args)
>>> 10
```


Högre ordningens funktioner

- Funktioner som hanterar funktionsobjekt
- Två typer
  - Funktionsobjekt skickade som parametrar till funktionen
  - Funktionsobjekt returneras av funktionen
- Python har inbyggdt stöd för HO-programmering
  - Varargs
  - Referens till funktionsobjektet via namnet
  - Inbyggda HO-funktioner för listor: map, filter, reduce
  - Lambda – för att göra enkla funktionsobjekt direkt på raden

Anonyma funktioner: lambda-uttryck

lambda <param-1>, <param-2>...<param-n>: <expression>

- Lambdauttryck kan användas för att skapa funktionsobjekt utan att införa ett namn
  - med lambda kan vi tilldela vanliga variabler funktionsobjekt
- Lambdauttryck är **uttryck** inte ett satsblock
- Användbara t ex när man vill skapa mindre funktionsobjekt tillfälligt eller för att skicka med i anrop

Exempel på lambdauttryck

```
>>> times = lambda x1, x2: x1 * x2
>>> times(3,4)
12
>>> plus = lambda x1, x2: x1 + x2
>>> operators = [times, plus]
>>> for fn in operators: fn(3,4)
12
7
```


Hur använda funktioner som objekt?

```
def sum_pow(max, exp):
    result = 0
    for i in xrange(1, max):
        result += pow(i,exp)
    return result
```

```
def sum_divide(max, div):
    result = 0
    for i in xrange(1, max):
        result += i / float(div)
    return result
```

```
print sum_pow(12, 6)
print sum_divide(12,6)
```

→

```
3749966
11.0
```

Hur kan vi "bryta ut" den gemensamma strukturen i sum_pow och sum_divide?

Funktioner som parametrar – högre ordningens funktioner

```
def sum_function(max, arg, sum_fn):
    result = 0
    for i in xrange(1, max):
        result += sum_fn(i, arg)
    return result
```

- Vanliga parametrar
- Anrop sker genom att ange parametern som funktionsnamn
- "Generisk funktion"
- Viktig konstruktion!

```
print sum_function(12, 6, pow)
div_float = lambda x,y: x / float(y)
print sum_function(12, 6, div_float)
```

→

```
3749966
11.0
```


okänd argumentlista – med varargs *

Scenario: vi samlar in funktionsobjekt, som ska appliceras senare. Den funktion som appliceras vet ofta inte vilka argument som krävs.

Lösning: varargs *

```
def run_all(fn_list, args_list):
    results = []
    for (fn,args) in zip(fn_list, args_list):
        result =fn (*args)
        results.append(result)
    return results
```

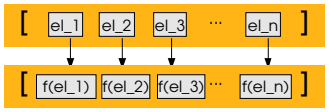
```
plus = lambda x,y: x + y
mult = lambda x,y: x * y
inv = lambda x: 1 / float(x)

print run_all([plus, mult, inv],
              [(2,3),(6,8),(13,)])
```

↓

```
[5, 48, 0.076923076923076927]
```

Map




```
>>> add_one = lambda x: x + 1
>>> map(add_one, [1,2,3,4])
[2, 3, 4, 5]

>>> pair_sum = lambda x,y: x + y
>>> map(pair_sum, [1,2,3,4], [4,3,2,1])
[5, 5, 5, 5]

>>> map(pair_sum,"aha", "boa")
['ab', 'ho', 'aa']
```

- Applicera given funktion på varje element i listan
- Generellt flera listor – iterera genom alla listor parallellt
- Antal argument till funktionen = antal listor
- Olika antal element ger None där element saknas
- Returnerar en lista

Exempel: map

```
def create_function_table(fn, min, max):
    return dict(map(fn, range(min, max)))

square = lambda x: (x, x ** 2)
print create_function_table(square, 1, 6)

cube = lambda x: (x, x ** 3)
print create_function_table(cube, 1, 6)
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
{1: 1, 2: 8, 3: 27, 4: 64, 5: 125}
```


Returnera funktionsobjekt

inc.py: skapar increment-funktioner dynamiskt, jmf ++-operatorn

```
def inc(number):
    return lambda x: x + number

inc_one = inc(1)
inc_two = inc(2)
inc_three = inc(3)

print inc_one(7)
print inc_two(7)
print inc_three(7)
```

```
8
9
10
```

- Vi kan skapa lambda-uttryck dynamiskt
- Funktioner som genererar funktioner
- Binda variabler och/eller kombinera funktioner dynamiskt

Summering

- Tal lagras binärt i en dator
  - Hexkod och oktal kod smidig notation
- Tecken kodas som tal och representeras binärt fysiskt
  - Unicode och UTF-8
- Modellering och abstraktion – nyckelbegrepp
- Datastrukturer med t ex strukturdelning för modellering
- ADTer för att gömma/abstrahera representationen
- Procedurell abstraktion för läsbarhet och generalitet
  - Högre ordningen funktioner och polymorfism ger generalitet