TDP002 – Imperativ programmering

Funktioner, felhantering och tabeller

Eric Elfving

Institutionen för datavetenskap (IDA)

Översikt

- Återblick underprogram
- Felhantering (igen)
- Mer om tabeller (`dict`)

Underprogram

- Underprogram används för att generalisera och upprepa kod.
- En funktion kan ta noll eller flera *parametrar* och returnera **ett** värde till anroparen.
- Definieras med `def`-satsen:

```
def funktionsnamn([parameter, …])
```

- Anropas från en annan funktion (eller modul) med *argument*

Underprogram

- Funktioner kan anropas med antingen positionsargument eller namngivna parametrar

```
def fun(a, b, c): pass
```

anrop	a	b	c
fun(4, 2, 5)	4	2	5
fun(b=2, c=5, a=6)	6	2	5
fun(2, c=3, b=5)	2	5	3
fun(b=2, 3, a=5)	SyntaxError: non-keyword arg after keyword arg		

Underprogram

- Underprogram kan ha defaultargument:

```
def print_line(n):
    for i in range(n):
        print('-', end='')
    print()

print_line(20)
```

```
def print_line(n=80):
    for i in range(n):
        print('-', end='')
    print()

print_line(20)
print_line()
```

Underprogram

- Om vi vill ha defaultargument måste de komma i slutet av parameterlistan:

```
def fun(a, b=4, c=2):
    return a+b+c
```

```
def fun(a, b=4, c):
    return a+b+c
```

SyntaxError: non-default argument follows default argument

- Detta för att interpretatorn annars får det svårt att veta om du vill ha defaultvärdet eller om det var nästa parameter du menade

Underprogram

- Vi har redan använt funktioner med defaultargument…
  - `print([object, ...], *, sep=' ', end='\n', file=sys.stdout)`
  - `open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True)`

Underprogram

- Defaultargument evalueras vid funktionsdefinitionen (en gång)

```
def f(val, lst=[]):
    lst.append(val)
    return lst

>>> f(1)
[1]
>>> f(2)
[1, 2]
>>> f(3)
[1, 2, 3]
```

```
def f(val, lst=None):
    if lst is None:
        lst = []
    lst.append(val)
    return lst

>>> f(1)
[1]
>>> f(2)
[2]
>>> f(3)
[3]
```

Felhantering

- Om något går fel kan en funktion kasta ett *undantag* (exception)
- Om det inte tas emot kommer programmet krascha
- Man tar emot ett undantag med ett try-except-block

```
try:
    int('två')
except ValueError:
    print('Fel vid omvandling!')
```

Felhantering

- De flesta inbyggda funktionerna som kastar undantag ger även ett felmeddelande.
- Det kan man komma åt enligt nedan

```
try:
    int('två')
except ValueError as Err:
    error_message = Err.args[0]
    print('Fel vid omvandling! (' + error_message +')')
```

Felhantering

- Om man inte vet vilket fel man väntar sig går det att ta emot *Exception*
- Vi kan även själva kasta undantag om något är fel

```
def err():
    #TODO: Skapa på riktigt
    raise NotImplementedError('Orkade inte skriva funktionen!')

try:
    err()
except Exception as e:
    print(repr(e))
```

Tabeller

- Består av nyckel-värde associationer
- Nycklarna måste vara *hashbara* (oförändlika), t.ex. numeriska värden och strängar
- Metoderna `keys` och `values` tar fram nycklarna respektive värdena separat

```
>>> tabell = {}
>>> tabell['namn'] = 'Kalle'
>>> a=tabell['hej']
Traceback (most recent call last):
  File "<pyshell#113>", line 1, in <module>
    a=tabell['hej']
KeyError: 'hej'
>>> tabell[1] = 3
>>> tabell.keys()
['hej', 1]
>>> tabell.values()
['Kalle', 3]
```

Tabeller

- Metoden `items` är bra att använda sig av för att iterera över en tabell

```
>>> for k,v in tabell.items():
        print(k,'=>',v)

1 => 3
name => Kalle
```

Linköpings universitet

expanding reality

www.liu.se