Introduktion til funktioner, moduler og scopes i Python

Denne artikel er fortsættelsen af "I gang med Python", som blevet publiceret her på sitet for et par måneder siden. Det anbefales at man starter med denne.

Skrevet den **12. Feb 2009** af **jensgram** I kategorien **Programmering / Andre** | ★★★☆☆

Funktioner og procedurer
I Python skelnes der ikke mellem funktioner og procedurer, da alle funktion defineret med "def" vil returnere en værdi. Selv hvis man ikke har inkluderet noget return-statement vil der returneres None. For funktioner gælder same navngivningsregler som for variabler. I korte træk; der skelnes mellem store og små bogstaver, der kan benyttes underscore, navnet må ikke starte med et tal!

Funktioner har til formål at indkapsle kode med henblik på genbrug. Oprettelse af funktioner i Python sker efter følgende opskrift:

Eksempel 1:

```
def <name>([arg1 [, arg2 [..., argN]]]):
    <statements>
```

Som ved kontrolstrukturer m.v. er det i Python indrykningen der afgrænser statements. Når man rykker "ud" igen betragtes funktionsdefinitionen som afsluttet. Funktioner skal endvidere deklareres før de kaldes, så det er vigtigt at have følgende struktur i sine Python-filer:

1) Imports - beskrives senere
2) Funktionsdeklarationer
3) Statements - altså resten af koden

Som allerede nævnt vil en funktion returnere None, hvis ikke andet er angivet. Ofte vil man dog benytte funktioner som funktioner (frem for procedurer), og her kommer return-keyword'et ind. Det er altid en god idé kun at benytte ér return-statement, men i princippet kan man have flere. Funktionen termineres når første return nås.

Eksempel 2:

```
def enProcedure():
    print 'test'
    print 'en linie til'

def enFunktion(a):
    return 'argument: ' + a

def enFunktionTil(a):
    if a == 1:
        return 'ja'
    else:
```

```
        return 'nej'

enProcedure()
print enFunktion('test')
print enFunktionTil('hej')

# Vil udskrive følgende:
# >>>
# test
# en linie til
# argument: test
# nej
```

Her ses desuden, at funktioner i Python er polymorfe - input kan være alle datatyper, da det ikke er specificeret nogen steder. Desuden ses brugen af argumenter til funktioner.

Funktionsargumenter
I eksempel 2 viste jeg hvordan man kan sende argumenter med til sine funktioner. Dette gør jeg ved at definere en eller flere variabler i def-linien. Python har en ret unik måde at matche disse argumenter på:

Eksempel 3:

```
# Via placering
def test1(a, b):
    return [a, b]

# Med defaults
def test2(a, b = 0):
    return [a, b]

# Unmatched position
def test3(*args):
    return args

# Unmatched keywords
def test4(**args):
    return args

# Kombineret
def test5(a, b = 5, *pos, **key):
    return [a, b, pos, key]

print test1(1, 2)
print test1(b = 2, a = 3)
print test2(12)
print test3(1, 2, 3)
print test4(a = 'noget', b = 12)
print test5(1, 2, 10, 11, minVar = 'test')

# Vil udskrive følgende:
# >>>
```

```
# [1, 2]
# [3, 2]
# [12, 0]
# (1, 2, 3)
# {'a': 'noget', 'b': 12}
# [1, 2, (10, 11), {'minVar': 'test'}]
```

Det ses altså, hvordan Python kan match på position, med keywords, med defaults, ukendt antal, ukendte keywords og en blanding af det hele. Alt i alt må det jo betegnes som meget modtageligt!

Scopes i Python
Jeg vil kort berøre begrebet scope, da det er ret vigtigt i forbindelse med funktioner. Scopes hænger sammen med namespaces. Eksempelvis kan en variabel deklareret inden i en funktion ikke tilgås udenfor denne. Fra en funktion kan man dog godt tilgå variabler m.v. udenfor denne, da Python benytter LEGB-rækkefølgen til at finde navne.

LEGB står for Local, Enclosed, Global og Built-in.

Local scope er variabler i funktioner samt argumenter. Her ledes først, men findes et navn ikke i dette namespace går Python en tak længere "ud".
Enclosed scope dækker over det scope der ligger udenfor selve funktionen men inden i en anden. Dette scope benyttes kun, hvis en funktion er deklareret inden i en anden funktion. Denne fremgangsmåde er nemlig tilladt i Python, men jeg er ikke begejstret for kode som denne:

Eksempel 4:

```
def matematik(x, y):
    def plus():
        return x + y


    sum = plus()
    return 'Resultat: ' + str(sum)
```

Eksemplet er helt gyldigt, og det ses endda hvordan x og y hentes fra enclosed scope. Jeg bryder mig bare ikke om idéen med funktioner i funktioner, men hvis du gør så brug det endelig.

Global scope dækker over navne "øverst" i filen - altså variabler, funktioner osv. der er sat i selve filen. Desuden kan man "hente" et navn ind i funktionens eget namespace med global-keyword'et. Som følge af LEGB-reglen er det ikke nødvendigt hvis man blot skal "læse" en værdi, men global skal benyttes, hvis man vil ændre referencen:

Eksempel 5:

```
var = 2

def func1():
    var = 3
    return var
```

```
def func2():
    global var
    var = 4
    return var

print 'var  :', var
print 'func1:', func1()
print 'var  :', var
print 'func2:', func2()
print 'var  :', var

# Vil udskrive følgende:
# >>>
# var  : 2
# func1: 3
# var  : 2
# func2: 4
# var  : 4
```

Her ses hvordan en global variabel deklareres. Herefter opretter func1() en variabel med samme navn (men i sit eget namespace, hvor den globale variabel ikke ændres). func2() derimod benytter global til at operere direkte på den global variabel, der således ændres.

Sidste scope er built-in, der dækker over Pythons indbyggede funktioner (str(), join(), open() osv.)

Import af moduler
Funktioner er ikke den eneste mulighed for at genbruge kode i Python. Til dette formål er moduler særdeles velegnede og som standard følger hele Python Standard Library med i Python-installationen. Det er en masse moduler, der kan importeres i ens programmer. For eksempel kan man importere modulet "re", hvis man får brug for at benytte regulære udtryk.

Syntaksen for import er som følger:

Eksempel 6:

```
import <module>[..., moduleN]
```

hvor de enkelte moduler vil importeres til deres egne namespaces. Eksempelvis kan man benytte regulære udtryk:

Eksempel 7:

```
import re

if re.match('^[0-9]$', '1'):
    print 'Det er et (og kun et) tal!'

# Udskriver: "Det er et (og kun et) tal!"
```

Man kan også give modulet et nyt navn med as-keyword'et eller benytter from til at hente til eget namespace

Eksempel 8:

```
import re as minRE
from re import match
from re import match as test

if minRE.match('^[0-9]$', '1'):
    print 'match med alias'

if match('[a-z]', 'c'):
    print 'match med i namespace'

if test('[0-9]', '6'):
    print 'match med alias i namespace'

# Vil udskrive følgende:
# >>>
# match med alias
# match med i namespace
# match med alias i namespace
```

Når man benytter from skal man være opmærksom på "name clash"; egne variabler, funktioner m.v. vil overskrives af det importerede!

Enhver Python-fil kan importeres, men hvis man importerer egne filer skal man huske på, at de skal være tilgængelige fra Pythons eget bibliotek eller via PYTHONPATH-bibliotekerne. Nemmest er det, hvis man importere filer fra samme bibliotek som den fil der indeholder import-kald.

Eksempel på iterativ vs. rekursiv funktion
Ovenstående skulle gerne have givet et indblik i oprettelse af funktioner i Python. Slutteligt vil jeg vise to eksempler på løsning af ét problem. Det skyldes, at der generelt er to hovedtilgange til løsninger på problemer:

1) Iterativ struktur
2) Rekursiv struktur

Den iterative struktur er baseret på brugen af løkker (for, while), mens den rekursive er funktionsbaseret. En løsning fra en rekursiv struktur er defineret ved en sluttilstand og en løsning på et mindre problem.

Problemet i dette eksempel er, at vi skal have lavet en funktion, der kan returnere der største fællesnævner for to heltal. Teoretisk findes den største fællesnævner ved at finde resten (R) fra en division af største tal (M) med mindste tal (N). Så længe R er større end nul sættes M = N og N = R, hvorefter udregningen gentages. Når R er nul er N den største fællesnævner (søg evt. på "Greatest Common Divisor" eller "Euclidean algoritm" på Google, hvis du vil vide mere).

Iterativt kan problemet løses med:

Eksempel 9:

```
def igcd(M, N):
    while M % N > 0:
        R = M % N
        M, N = N, R
    return N

print igcd(15, 6) # 3
print igcd(270, 18) # 18
```

Her benyttes en while-løkke til at teste om resten er større end nul. Er den ikke det (altså er den 0) returneres N.

Rekursivt kan det løses som:

Eksempel 10:

```
def rgcd(M, N):
    R = M % N
    if R > 0:
        N = rgcd(N, R)
    return N

print rgcd(15, 6) # 3
print rgcd(270, 18) # 18
```

Her benyttes et if-statement til at teste og N returneres hvis R er lig nul. kaldes rgcd() med M = N og N = R.

Afslutning
Nu er du forhåbentlig blevet en anelse klogere på funktioner, moduler og scopes i Python. Hvis ikke er det beklageligt, men feedback er altid velkomment.

- Jens Gram, http://www.jensgram.dk/

Relaterede artikler og ressourcer
- I gang med Python (http://www.eksperten.dk/artikler/483)

Change log:

2004/02/12: Første version publiceret

Kommentar af simonvalter d. 23. Dec 2004 | 1

værd at læse hvis man er begynder.

Kommentar af soreno d. 05. Dec 2004 | 2

Jeg synes der mangler en motivation for at vælge python frem for andre sprog (f.eks. en sammenligning med de mest almindelige imperative sprog og nogle af de funktionsorienterede sprog.).