Teil IV

Funktionen und Module

Wozu Funktionen?

Wo sie schon verwendet wurden:

- Mathematische Funktionen: `sqrt(s)`, `sin(s)`, `exp(x)`, `...`
- Methoden von Sequenzen: `s.islower()`, `l.append(x)`, `...`
- Erzeugung von Listen: `range(x)`
- Typ und Identität: `type(x)`, `id(x)`
- Alle bisherigen haben keinen oder einen Übergabewert und einen Rückgabewert

Vorteile

- Zerlegung eines großen Problems in viele kleine
- Zusammenfassung von Anweisungsblöcken
- Klare Abgrenzung von Funktionalitäten
- Vermeidung von Code-Duplikation
- Höhere Code-Lesbarkeit (Wenn man statt `sin(x)` jedesmal den kompletten Code ...)
- ...

Syntax von Funktionen

- Funktionen werden definiert mit dem Schlüsselwort `def`
- Nach Funktionsname und Parameterliste kommt ein Doppelpunkt
- Der darauf folgende Block (Einrückung!) wird bei Aufruf der Funktion ausgeführt
- Ausführung bis zum Ende des Blocks oder bis zu einen `return`
- Rückgabewert kann beliebiger Typ sein (Zahl, Liste, String, ...)
- Ohne `return` oder ohne Wert wird `None` zurückgegeben

```
>>> def hi():
>>>     print "Hello␣World!"
>>>
>>> hi()
Hello World
>>> a = hi()
Hello World
>>> type(a)
<type 'NoneType'>
```

Funktionen sind Objekte

```
>>> def faculty(n):
>>>     fac = 1
>>>     for i in range(2, n+1):
>>>         fac = fac*i
>>>     return fac
>>>
>>> faculty(3)
6
>>> f = faculty
>>> f(4)
24
>>> type(f)
<type 'function'>
```

Funktionsparameter

- Parameter stehen in runden Klammern hinter dem Funktionsnamen
- Formalparameter: Bei der Definition der Funktion
- Aktualparameter: Folge von Ausdrücken beim Aufruf der Funktion
- Beliebig viele Parameter möglich, normalerweise gleich viele Formal- und Aktualparameter

```
>>> def potenziere(basis, exponent):
>>>     return basis**exponent
>>>
>>> potenziere(2,10)
1024
```

Namensparameter

- Bisher: Folge von Aktualparametern gemäß Reihenfolge der Formalparameter (Positionsparameter)
- Durch Angabe von Folge `Formalparameter=Aktualparameter` bei Aufruf beliebige Reihenfolge möglich (Namensparameter)

```
>>> x = potenziere(basis=2, exponent=10)
>>> x = potenziere(exponent=10, basis=2)
```

- Mischung von Positions- und Namensparameter möglich
- Erst Folge von Positions- dann die verbleibenden Namensparameter

```
>>> x = potenziere(2, exponent=10)
>>> x = potenziere(basis=2, 10) # Fehler
>>> x = potenziere(2, basis=2) # Fehler
```

Standardwerte für Parameter

- Namensparameter relativ nutzlos, wenn alle angegeben werden müssen
- Formalparameter können mit Standardwerten belegt werden
- Zuerst alle Formalparameter ohne Standardwerte, dann die mit

```
def potenziere(basis=2, exponent):   # Falsch
def potenziere(basis, exponent=10):
def potenziere(basis=2, exponent=10):
```

```
>>> potenziere(2,10) # (2,10)
>>> potenziere(2) # (2,2)
>>> potenziere(exponent=7) # (2,7)
```

- Für diese Funktion sind Standardwerte also relativ nutzlos
- Bei vielen Parametern kann es hilfreich sein

Rückgabewerte

- Bei mehreren Rückgabewerten wird ein Tupel zurückgegeben

```
>>> def return_two():
>>>     return "x", "y"
>>>
>>> h = return two();   # h = ('x', 'y')
>>> (a,b) = return_two() # a = 'x', b = 'y'
```

- Eine Liste oder explizites Tupel ist ein einzelner Wert

```
>>> def primfaktoren(x):
>>>   l=[]; n=2
>>>   while n <= x:
>>>     while x%n==0:
>>>       x /= n
>>>       l.append(n)
>>>     n += 1
>>>   return l
```

Gültigkeitsbereich

Funktionen

- Erinnerung: Python ist eine interpretierte Sprache
- Funktionen müssen definiert sein, bevor sie aufgerufen werden können

Variablen

- Unterscheidung zwischen lokalen und globalen Variablen
- Eine Variable, die in einer Funktion definiert (an einen Wert gebunden) wird, ist außerhalb nicht sichtbar
- Grundregel: GLOBALE VARIABLEN VERMEIDEN!!!
- Variablen, die in einer Funktion definiert oder links in einer Zuweisung verwendet werden, sind lokal
- Globale Variablen können gelesen werden
- Um globale Variablen in einer Funktion zu schreiben wird das Schüsselwort `global` verwendet

Beispiele

```
>>> def zaehle(n):
>>>     global anzahl
>>>     anzahl += 1
>>>
>>> anzahl = 0
>>> zaehle(); zaehle(); zaehle()
>>> anzahl
3
```

```
>>> def no_effect():
>>>     xyz = global**2 # Nochmal: ganz schlechter Stil
>>>
>>> global = 9
>>> no_effect()
>>> xyz
NameError: name 'xyz' is not defined
```

Docstrings

- Ein Grund für Funktionen: Funktionalität kapseln
- Jemand, der die Funktion nicht geschrieben hat, soll sie verwenden können
- Dazu ist Dokumentation nötig
- Ein String (ein- oder mehrzeilig) nach dem header ist dafür vorgesehen
- Python ignoriert den String (Ausdruck ohne Zuweisung)
- Das Hilfesystem (und der Code-Leser) kann den String verwenden
- Nicht nur für Funktionen, sondern auch für Klassen, Module, ...

```
>>> def blubb():
>>>     "Unsinnige Funktion, die 'blubb' ausgibt"
>>>     print "blubb"
>>> blubb()
blubb
>>> help(blubb)
```

Call by Reference vs. Call by value

- Call by Reference: „Adresse“(id) wird übergeben
- Call by Value: Wert der Variablen wird übergeben
- In Python immer Call by Reference
- Allerdings: Zuweisung ändert nur Referenz innerhalb der Funktion

```
>>> def add_one(x):
>>>     x = x + 1
>>>
>>> x = 3
>>> add_one(x)
>>> x
3
```

Call by Reference vs. Call by value (2)

- Nach der Zuweisung zeigt die lokale Variable auf ein neues Objekt
- Mit Methoden kann das bisherige Objekt verändert werden

```
>>> def append1(l):
>>>   l = l + [4]
>>>
>>> def append2(l):
>>>   l.append(4)
>>>
>>> l = [1,2,3]
>>> append1(l); l
[1,2,3]
>>> append2(l); l
[1,2,3,4]
```

Anonyme Funktionen

- Normale Funktionen müssen separat definiert werden
- Sie bekommen einen Namen, mit dem auf sie zugegriffen wird

```
>>> def inc(i):
>>>     return i+1
>>> inc(3)
4
```

- Namenlose/Anonyme Funktionen erhalten keinen
- Sie können an beliebigen Stellen in den Code eingebaut werden

```
>>> inc = lambda(i): i+1
>>> inc(3)
4
```

map

- `map` wendet eine Funktion auf alle Elemente einer Liste an
- Die Funktion kann natürlich auch anonym sein

```
>>> l = range(10)
>>> map(lambda x: x*x+1, l)
[1, 2, 5, 10, 17, 26, 37, 50, 65, 82]
```

- In diesem Fall wäre das sogar noch kürzer mit list comprehensions (nächste Folie) gegangen

filter

- `filter` wendet ebenfalls eine Funktion auf alle Elemente einer Liste an
- Diejenigen Elemente, für die die Funktion `True` zurückgibt, werden zurückgegeben.

```
>>> filter(lambda x: x%2==0, l)
[0, 2, 4, 6, 8]
```

List Comprehension

- Einfache Listen lassen sich z.B. mit `range` erzeugen
- Für komplexere Listen gibt es list comprehension

```
>>> b = [i**2 for i in range(0,10) it i%2==0]
>>> b
[0, 4, 16, 36, 64]
```

- Generelle Syntax:

```
[expression for item1 in iterable1
            for item2 in iterable2
            ...
            for itemN in iterableN
            if condition]
```

Beispiel: Numerische Integration (Quadratur)

Berechnung des Integrals

$$F_1(f, a, b) := \int_a^b f(x)\, dx$$

Rechtecksregel

- Bestimmung des Funktionswertes in der Intervallmitte
- Multiplikation mit der Intervallbreite
- $F_1 \approx R := (b - a) \cdot f(\frac{a+b}{2})$
- Polynom 0. Ordnung

Trapezregel

- Bestimmung des Funktionswertes an beiden Rändern
- Multiplikation des Mittelwerts mit der Intervallbreite
- $F_1 \approx T := (b - a) \cdot \frac{f(a)+f(b)}{2}$
- Polynom 1. Ordnung (Fläche entspricht Trapez)

Simpsonregel

- Anderer Name: Keplersche Fassregel
- Bestimmung des Funktionswertes an beiden Rändern und in der Mitte
- Polynom 2. Ordnung durch diese drei Punkte
- Fläche unter dem Polynom wird berechnet
- $F_1 \approx S := (b-a) \cdot \frac{f(a)+4 \cdot f(\frac{a+b}{2})+f(b)}{2}$

Quadraturfehler

$$|T - F_1| \leq \frac{1}{12} \cdot \sup_{x \in [a,b]} |f''(x)| \cdot (b-a)^3$$

$$|S - F_1| \leq \frac{1}{2880} \cdot \sup_{x \in [a,b]} |f^{(4)}(x)| \cdot (b-a)^5$$

Summenregeln

- Bei großen Intervallen wird auch der Fehler sehr groß
- $\Rightarrow$ Unterteilung in n kleine Intervalle
- Jedes Teilintervall hat Länge $(b-a)/h$

Trapezsumme

$$TS := h \cdot \left[\frac{f(a)}{2} + \sum_{i=1}^{n-1} f(a+ih) + \frac{f(b)}{2} \right]$$

Simpsonsumme

$$\begin{aligned} SS \quad := \quad & \frac{h}{6} \left[f(a) + 4f\left(a+\frac{h}{2}\right) + 2f(a+h) + 4f\left(a+\frac{3h}{2}\right) + \ldots \right. \\ & \left. + 4f\left(b-\frac{h}{2}\right) + f(b) \right] \end{aligned}$$

Fehler der Trapezsumme

- In jedem Teilintervall ist der Fehler

$$\leq \frac{1}{12} \cdot \sup_{x \in [a,b]} |f''(x)| \cdot h^3$$

- Bei $n = (b-a)/h$ Intervallen ist damit der Gesamtfehler

$$|TS - F_1| \leq \frac{1}{12} \cdot \sup_{x \in [a,b]} |f''(x)| \cdot (b-a) \cdot h^2$$

- Verdopplung des Rechenaufwands (Halbierung von h) reduziert den maximalen Fehler um den Faktor 4

Fehler der Simpsonsumme

- Analog zur Trapessumme ergibt sich

$$|SS - F_1| \leq \frac{1}{2880} \cdot \sup_{x \in [a,b]} |f^{(4)}(x)| \cdot (b-a) \cdot h^4$$

- Verdopplung des Rechenaufwands (Halbierung von h) reduziert den maximalen Fehler um den Faktor 16

Implementierung der Trapezsumme

```
def trapezsumme(f, a, b, n):
  h = (b-a)/n
  sum = (f(a) + f(b))/2
  for i in range(1,n):
    sum = sum + f(a + h*i)
  return sum*h
```

Funktion zum Test

- Funktion, die analytisch leicht zu integrieren
- $\Rightarrow$ Fehler lässt sich leicht berechnen

```
import math
def f(x):
  return math.sin(math.pi*x)
def f_int(x):
  return -math.cos(math.pi*x)/math.pi
```

Berechnung mit unterschiedlicher Genauigkeit

- $n = 2^0 ... n^6$

```
a_bsp = 0.
b_bsp = 1.
exakt = f_int(b_bsp) - f_int(a_bsp)
for i in range(1,7):
    n = 2**i
    t = trapezsumme(a_bsp, b_bsp, n)
    print '%4d␣%-12.6g␣%-12.6g' % (n, t, exakt-t)
```

N	Näherung	Fehler
2	0.5	0.13662
4	0.603553	0.0330664
8	0.628417	0.00820234
16	0.634573	0.00204662
32	0.636108	0.000511409
64	0.636492	0.000127837

Module

- Funktionalität zusammenfassen in einer separaten `.py` Datei
- Importieren Bibliotheksmodul (Schon verwendet: math, time, ...)
- Beispiel (`tools.py`):

```
"""This␣module␣provides␣some␣helper␣tools.
Try␣it."""

counter = 42

def readfile(fname):
    "Read␣text␣file.␣Returns␣list␣of␣lines"
    fd = open(fname, 'r')
    data = fd.readlines()
    fd.close()
    return data

def do_nothing():
    "Do␣really␣nothing"
    pass
```

Importieren

- Modul importieren

```
import tools
tools.do_nothing()
print tools.counter
```

- Modul unter neuem Namen importieren

```
import tools as t
t.do_nothing()
```

- Einzelne Funktionen direkt importieren

```
from tools import do_nothing, readfile
from tools import counter as cntr
do_nothing()
print cntr
```

Import beeinflussen

- Alle Symbole in den Namensraum importieren

```
from tools import *
do_nothing()
print counter
```

- Module können bestimmen, welche Symbole mit `from module import *` importiert werden:

```
# module tools.py
__all__ = ['readfile', 'counter']
```

Die Funktion `do_nothing()` ist nach `import *` nicht bekannt!

Funktionalität abfragen

```
dir(tools)
dir(math)
```

Hilfe zu Modulen und Funktionen

- Docstrings abfragen mit `help`

```
import tools
help(tools)
help(tools.do_nothing)
```

Ausführbares Programm als Modul?

- `tools.py` soll sowohl Bibliothek sein, als auch selbst ausführbar

```
# tools.py
...
if __name__ == '__main__':
    print "tools.py executed"
else:
    print "tools.py imported as module"
```

Module debuggen

- Wenn ein Modul geändert wird, werden Änderungen nur wirksam, wenn Python neu gestartet wird, oder `reload` (Python<3.0) verwendet wird

```
reload(tools)
```

Pfad für Module

- Python sucht im Suchpfad und im aktuellen Verzeichnis
- Suchpfad kann erweitert werden

```
import sys
sys.path.append("folder/to/module")
import ...
```

Pakete

- Module können gruppiert werden
- Hierbei ist die Verzeichnisstruktur wichtig

```
tools/
    __init__.py      # Inhalt für "import tools"
    files.py         # für "import tools.files"
    graphics.py      # für "import tools.graphics"
    stringtools/
        __init__.py  # für "import tools.stringtools"
        ... weitere Verschachtelung möglich
```

- Wenn `from tools import *` Untermodule enthalten soll, muss `tools/__init__.py` folgende Zeile enthalten

```
__all__ = ["files", "graphics"]
```