Funktionen

Funktionen sind ein wichtiger Baustein eines Computerprogramms. Sie erlauben eine bessere Gliederung des Codes und erleichtern die Wiederverwendung von Code-Stücken.

Grundlagen

Um eine Funktion zu erstellen wird das *def* keyword benötigt, anschließend kommt der Funktionsname, und dahinter eine Klammer, in die die Funktionsargumente geschrieben werden (falls die Funktion welche benötigt).

```
def quadrat(x):
    return x*x
```

In dem oberen Beispiel hat die Funktion einen Rückgabewert. Das muss aber auch nicht zwingend sein:

```
def hello_world():
    print "Hello World"
```

Man kann den Funktionsargumente Defaultwerte geben, die benutzt werden, falls die Funktion ohne Parameter aufgerufen wird:

```
def summe(a=0, b=0, c=0):
    return a + b + c

print summe()
```

Ausgabe:

```
0
```

Es ginge aber z.B. auch:

```
summe(1, 2) # a und b werden auf 1 und 2 gesetzt, c hat den Defaultwert 0

summe(a=100) # a wird auf 100 gesetzt, b und c haben den Defaultwert 0

summe(3, c=99.912) # a wird auf 3 und c auf 99.912 gesetzt, b hat den Defaultwert 0
```

Darüberhinaus ist es möglich, eine Funktion zu schreiben, die beliebig viele Argumente akzeptiert:

```
def summe(*args):
    total = 0
    for value in args:
        total += value
    return total
```

```
print summe(1)
print summe(1, 2, 3)
print summe(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Ausgabe:

```
1
6
55
```

Alle Argumente, die *summe* übergeben werden, werden in einem Tupel namens *args* gespeichert. Über diesen können wir dann iterieren.

Rekursive Funktionen

Falls Sie rekursive Funktionen nicht verstehen, lesen Sie diesen Satz bitte nochmal.

Scherz beiseite, rekursive Funktionen sind Funktionen, die ein Problem lösen, indem sie sich selbst nochmal aufrufen.

Das bekannsteste Beispiel ist vermutlich die Fakultätsfunktion: Wie die meisten vermutlich noch aus der Schulzeit (oder aus der Mathevorlesung) wissen, ist die Fakultät einer natürlichen Zahl n definiert als:

```
n! = n * (n-1) * (n-2) * ... * 1
```

Obige Formel können wir aber auch folgendermaßen schreiben:

```
n! = n * (n-1)!
```

In dieser Schreibweise sagt uns die Formel also, dass die Fakultät von n einfach n mal die Fakultät der Vorgängerzahl von n ist.

Hier fehlt allerdings noch eine Kleinigkeit: wir brauchen einen Rekursionsanker, der uns sagt, wann wir aufhören können zu rechnen. In diesem Fall ist dieser Anker die Tatsache, dass $0! = 1$.

Als Programm geschrieben, sieht das ganze so aus:

```
def fakultaet(n):
    if n == 0:
        return 1
    else:
        return n * fakultaet(n-1)
```

Klassen und Objekte

Aus Zeitgründen wird dieses Thema nur kurz angeschnitten werden, auch wenn man eigentlich sehr viel mehr Zeit damit verbringen könnte.

Wir hatten zuvor gesehen, dass wir mit Funktionen gewissermaßen unseren Code etwas aufräumen können. Code der häufiger verwendet wird, wird als Funktion definiert, und später nur noch über den Funktionsnamen aufgerufen.

Ebenso können wir nun Variablen **und** Funktionen zusammenfassen. Das geht über ein Objekt. Die Klasse ist dabei die Blaupause, aus der ein Objekt erstellt wird. Aus einer Klasse können beliebig viele Objekte erstellt werden.

Eine Klasse wird folgendermaßen in Python definiert:

```
class MeineKlasse:
    var1 = ...
    var2 = ...
    ...
    def __init__(self, p1, p2, p3, ...):
        ...
    def methode1(self, q1, q2, ...)
        ...
    def methode2(self, ...)
        ...
    ...
```

var1 und *var2* sind hier Objektvariablen, *methode1* und *methode2* Methoden, d.h. Funktionen, die jedes Objekt dieser Klasse besitzen wird.
Die Methode *__init__* ist eine besondere Methode. Sie wird beim Erstellen eines Objekts aufgerufen.

Ein Beispiel aus der Physik: wir wollen ein freies Teilchen in zwei Dimensionen simulieren, das durch den Raum fliegt. Wir können es über Position und Geschwindigkeit beschreiben. Darüberhinaus wollen wir ihm eine *move* Methode geben, die als Parameter einen Zeitschritt akzeptiert, und das Teilchen in dieser Zeit weiterbewegt.

```
class Particle:
    def __init__(self, x, y, v_x, v_y):
        self.x, self.y = x, y # hier wird die x, y Position des Teilchens festgelegt
        self.v_x, self.v_y = v_x, v_y # hier wird die Geschwindigkeit festgelegt
    def move(self, delta_t):
        self.x += self.v_x * delta_t # Teilchen wird in x-Richtung bewegt
        self.y += self.v_y * delta_t # Teilchen wird in y-Richtung bewegt
```

Wann immer wir auf eine Objektvariable zugreifen möchten, müssen wir dies mit einem vorangestellten *self.* kenntlich machen.

Wir können nun folgendermaßen ein Objekt erstellen:

```
mein_teilchen = Particle(0, 0, 1, 3)  # die Parameter, die wir hier angeben, werden an die _
print mein_teilchen.x, mein_teilchen.y
for i in range(10):
    mein_teilchen.move(0.1) # ruft die move Methode mit delta_t = 0.1 auf
    print mein_teilchen.x, mein_teilchen.y
```

In der ersten Zeile haben wir nun ein Objekt der Klasse *Particle* erstellt und es *mein_teilchen* genannt. Wir können nun auf die Methoden und Objektvariablen unseres Objekts zugreifen, indem wir hinter dem Objektnamen einen Punkt und dann den Objektvariablennamen bzw. Methodennamen schreiben.
Es ist so sogar möglich, neue Objektvariablen auf diese Weise hinzuzufügen nachdem das Objekt schon erstellt wurde.

Funktionen als Objekte

Diese Tatsache, mag etwas verwunderlich erscheinen, aber tatsächlich sind in Python auch Funktionen Objekte.

So könnten wir z.B. eine Funktion schreiben, die eine andere Funktion als Argument bekommt, und diese dann aufruft:

```
def f(x):
    return x*x

def werte_funktionen_aus(irgendeine_funktion):
    for i in range(10):
        print irgendeine_funktion(i)

werte_funktionen_aus(f)
```

Dieser Code würde also die Funktion f(x) auswerten an den Stellen 0 bis 9.

Andersherum können wir aber auch Funktionen definieren, die wiederum Funktionen zurückgeben:

```
def polynom_creator(degree):
    def polynom(x):
        total = 0
        for i in range(degree, -1, -1):
            total += x**i
        return total
    return polynom # hier wird das Funktionsobjekt zurückgegeben
```

Diese Funktion gibt uns ein Polynom der Form

```
p(x) = x^n + x^{n-1} + ... + 1
```

zurück.

```
polynom_dritten_grades = polynom_creator(2) # hier erstellen wir ein Polynom zweiten Grades
polynom_zweiten_grades = polynom_creator(3) # und hier eins dritten Grades
print polynom_zweiten_grades(5)
print polynom_dritten_grades(5)
```

Ausgabe:

```
31
156
```

Grundlegende Verständnisfragen:

- Was gibt die Funktion *polynom_creator* zurück?
- Was gibt die Funktion *polynom* zurück?

Scope – Der Geltungsbereich von Variablen

Was passiert in folgendem Beispiel?

```
x = 0

def f():
    x = 7
    print "x wurde gesetzt auf", x

print "vor Ausführung von f: x =", x
f()
print "nach Ausführung von f: x =", x
```

Es sieht so aus, als würde die Funktion f unsere zuvor deklarierte Variable x ändern. Tatsächlich ist die Ausgabe aber

```
vor Ausführung von f: x = 0
x wurde gesetzt auf 7
nach Ausführung von f: x = 0
```

Der Grund für dieses Verhalten ist, dass Python dafür sorgt, dass Funktionen nicht aus Versehen globale Variablen (d.h. Variablen, die auf oberster Ebene definiert wurden) ändert. Damit die globale Variable x nicht überschrieben wird, legt Python eine weitere Variable x an, die nur innerhalb der Funktion f Gültigkeit hat. Nach Verlassen der Funktion existiert diese nicht mehr, und es existiert nur noch das globale x.

Ein anderes Beispiel:

```
x = 0

def f():
```

```
    print "x hat den Wert:", x

f()
```

In diesem Beispiel haben wir innerhalb von f keine Variable x definiert. Dennoch funktioniert der Code. Python merkt in diesem Fall nämlich, dass keine lokale Variable x existiert und schaut deshalb nach, ob es eine globale Variable gibt.

Aber Vorsicht: sobald irgendwo in der Funktion auch eine Variable x auftaucht, existiert diese lokal, und Python wird nicht mehr im globalen Scope nach x suchen:

```
x = 0

def f():
    if False:
        x = 7
    print "x hat den Wert:", x

f()
```

Obwohl der Code nach der if-Bedingung niemals ausgeführt wird, sorgt er dafür, dass x nun lokal existiert. Allerdings wird dem lokalen x nie ein Wert zugewiesen, weshalb das obige Beispiel mit einer Fehlermeldung endet.

Funktions-Wrapper

Dass wir Funktionen wie Objekte anderen Funktionen übergeben dürfen, eröffnet uns nun ganz neue Möglichkeiten. So können wir nun eine Funktion schreiben, die eine andere Funktion als Argument nimmt und daraus eine neue Funktion macht!

Beispiel:

```
def wrapper(f):
    def f_enhanced():
        print "Am Anfang..."
        f()
        print "und am Ende passiert was neues!"
    return f_enhanced

def f():
    print "Hallo Welt"

f()
f_neu = wrapper(f)
f_neu()
```

Die Funktion *wrapper* nimmt also eine Funktion, und macht daraus eine neue Funktion mit erweiterter Funktionalität. Natürlich hätten wir im obigen Beispiel auch einfach die Funktion f umschreiben können, allerdings wird das spätestens dann anstrengend, wenn wir das für mehrere Funktionen machen wollen.

Ein weiteres Beispiel wäre z.B. ein wrapper, der Funktionen dahingehend erweitert, dass sie ihre Argumente auf bestimmte Kriterien prüfen:

```
def check_args_nonnegative(f):
    def f_enhanced(*args):
        for arg in args:
            if arg < 0:
                raise ValueError("Bitte nur Argumente größer Null eingeben!")
        return f(*args)
    return f_enhanced
```

Zu diesem Beispiel sind noch ein paar Worte zu sagen: wir sehen zum einen, dass f_enhanced mit *args als Argument definiert wurde. D.h. sie nimmt beliebig viele Argumente. Das ist notwendig, da f_enhanced ja mit vielen verschiedenen Funktionen soll, die unterschiedlich viele Funktionsargumente erwarten.

Bevor nun die ursprüngliche Funktion f aufgerufen wird, wird zuerst geprüft, ob eines der gegebenen Argumente kleiner Null ist. In diesem Fall wird ein Fehler ausgelöst und die Funktion beendet sich.

Diesen 'Wrapper' können wir jetzt beispielsweise auf unsere Fakultätsfunktion anwenden:

```
fak = check_args_nonnegative(fak) # wir speichern die neue Funktion wieder unter dem selben
fak(5) # ok
fak(3) # ok
fak(0) # ok
fak(-1) # ValueError!
```

Alternativ können wir anstelle von 'fak = check_args_nonnegative(fak)' auch einen sogenannten Decorator verwenden. Diesen müssen wir direkt über die Funktionsdefinition schreiben:

```
@check_args_nonnegative
def fak(n):
    if n == 0:
        return 1
    else:
        return n * fak(n-1)
```

Die Funktion *fak* hat dann automatisch die erweiterte Funktionalität.