

6 Rekursive Grundstrukturen

Rekursion, auch *Rekurrenz* oder *Rekursivität*, bedeutet Selbstbezüglichkeit (von lateinisch *recurrere* = zurücklaufen; eng. *recursion*). Sie tritt immer dann auf, wenn etwas auf sich selbst verweist. Ein rekursives Element muss nicht immer **direkt** auf sich selbst verweisen (**direkte Rekursion**), eine Rekursion kann auch über mehrere Zwischenschritte entstehen. Rekursion kann dazu führen, dass „merkwürdige Schleifen“ entstehen. So ist z.B. der Satz „Dieser Satz ist unwahr“ rekursiv, da er von sich selber spricht. Eine etwas subtilere Form der Rekursion (**indirekte Rekursion**) kann auftreten, wenn zwei Dinge gegenseitig aufeinander verweisen. Ein Beispiel sind die beiden Sätze: „Der folgende Satz ist wahr“, „Der vorhergehende Satz ist nicht wahr“.

Im Zusammenhang mit der Definition der Rekursion werden des Öfteren folgende Sätze genannt:

„Um Rekursion zu verstehen, muss man erst einmal Rekursion verstehen.“
 „Kürzeste Definition für Rekursion: siehe Rekursion.“

Rekursion ist ein allgemeines Prinzip (wie die Iteration) zur Lösung von Problemen. In vielen Fällen ist die Rekursion eine von mehreren möglichen Problemlösungsstrategien, sie führt oft zu „eleganten“ Lösungen.

Als **Rekursion** bezeichnet man den Aufruf oder die Definition einer Funktion (hier wird Funktion sowohl wie in der Mathematik im Sinne von Abbildung gebraucht als auch im Sinne einer speziellen Prozedur, die einen Wert zurückgibt, siehe unten) durch sich selbst. Ohne geeignete Abbruchbedingung geraten solche rückbezüglichen Aufrufe in einen so genannten infiniten Regress (umgangssprachlich auch oft Endlosschleife genannt, siehe auch in der Vorlesung 6).

Zur Vermeidung von infinitem Regress insbesondere in Programmen bedient man sich der semantischen Verifikation von rekursiven Funktionen. Der Beweis, dass kein infiniter Regress vorliegt, wird dann zumeist mittels einer Schleifeninvariante geführt. Dieser Beweis ist allerdings nicht immer möglich (z.B. bei dem so genannten Halteproblem, das später in der Theoretischen Informatik behandelt wird).

Wir benutzen im Weiteren Funktionen auf natürlichen Zahlen, um uns das Prinzip klarzumachen. Rekursion oder rekursive Definitionen sind allerdings grundsätzlich nicht auf natürliche Zahlen beschränkt.

Die **Grundidee der rekursiven Definition** einer Funktion f ist folgende: Der Funktionswert $f(n+1)$ einer Funktion $f: \mathbf{N}_0 \rightarrow \mathbf{N}_0$ ergibt sich durch Verknüpfung bereits vorher berechneter Werte $f(n), f(n-1), \dots$. Falls außerdem die Funktionswerte von f für hinreichend viele Startargumente bekannt sind, kann jeder Funktionswert von f berechnet werden. Das heißt im Klartext: Bei einer rekursiven Definition einer Funktion f ruft sich die Funktion so oft selber auf, bis ein vorgegebenes Argument (meistens 0) erreicht ist, so dass die Funktion terminiert (abbricht).

Die Definition von rekursiv festgelegten Funktionen ist eine grundsätzliche Vorgehensweise in der funktionalen Programmierung. Ausgehend von einigen gegebenen Funktionen (wie z. B. der Summen-Funktion) werden neue Funktionen definiert. Mit diesen können weitere Funktionen definiert werden, usw.

Ein Spezialfall der Rekursion ist die **primitive Rekursion**, die durch eine Iteration ersetzt werden kann. Bei einer solchen Rekursion enthält der Aufruf-Baum keine Verzweigungen, das

heißt er ist eigentlich eine Aufruf-Kette: das ist immer dann der Fall, wenn eine rekursive Funktion sich selbst jeweils nur einmal aufruft, insbesondere am Anfang (*Head Recursion*) oder nur am Ende (*Tail Recursion*) der Funktion. Umgekehrt kann jede Iteration durch eine primitive Rekursion ersetzt werden, ohne dass sich dabei die Komplexität des Algorithmus ändert.

Betrachten wir zunächst ein Beispiel:

Die Funktion sum : berechnet die Summe der ersten n Zahlen.

Die Funktion $\text{sum}: \mathbf{N}_0 \rightarrow \mathbf{N}_0$ sei definiert durch: $\text{sum}(n) = 0 + 1 + 2 + \dots + n$, berechnet also die Summe der ersten n Zahlen.

Anders ausgedrückt: $\text{sum}(n) = \text{sum}(n-1) + n$ (Rekursionsschritt)

Das heißt also, die Summe der ersten n Zahlen lässt sich berechnen, indem man die Summe der ersten $n - 1$ Zahlen berechnet und dazu die Zahl n addiert.

Damit die Funktion terminiert, legt man hier für $\text{sum}(0) = 0$ (Rekursionsanfang) fest.

Mit diesen Angaben lässt sich eine rekursive Definition angeben, die eine beliebige (hier: natürliche) Zahl x berechnet. Die Definition lautet also:

$$\text{sum}(n) = \begin{cases} 0 & \text{falls } n = 0 \text{ (Rekursionsanfang)} \\ \text{sum}(n-1) + n & \text{falls } n \geq 1 \text{ (Rekursionsschritt)} \end{cases}$$

Man errechnet dann

$$\begin{aligned} \text{sum}(3) &= \text{sum}(2) + 3 \\ &= \text{sum}(1) + 2 + 3 \\ &= \text{sum}(0) + 1 + 2 + 3 \\ &= 0 + 1 + 2 + 3 \\ &= 6 \end{aligned}$$

Im Fall von primitiv-rekursiven Funktionen steht es dem Programmierer frei, eine iterative oder eine rekursive Implementation zu wählen. Dabei ist die rekursive Umsetzung meist „eleganter“, während die iterative Umsetzung effizienter ist (insbesondere weil der Overhead für den wiederholten Funktionsaufruf entfällt). Betrachten wir ein weiteres Beispiel bei dem eine rekursive Lösung einer iterativen Lösung gegenübergestellt wird: Berechnung der Fakultät einer Zahl (Gamma-Funktion, wenn auf den reellen Zahlen definiert):

Für alle natürlichen Zahlen $n \in \mathbf{N}$

$$n! = \prod_{k=1}^n k = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$$

also:

- $3! = 1 \cdot 2 \cdot 3 = 6$
- $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$

- häufig definiert man zusätzlich $0! = 1$ (hier liegt das leere Produkt vor)

Betrachten wir auch hierzu ein Beispiel: Der rekursiven Variante wird eine iterative Variante gegenübergestellt.

n sei die Zahl, deren Fakultät berechnet werden soll

```
fakultät_rekursiv (n)
if n <= 1
then return 1
else return n * fakultät_rekursiv (n-1)
```

Die Rekursion kommt in Zeile 3 zum Ausdruck, wo die Funktion sich selbst mit einem um 1 verringerten Argument aufruft.

Eine iterative Lösung sieht wie folgt aus:

```
fakultät_iterativ(n)
fakultät := 1
faktor := 2
while faktor <= n
    fakultät := fakultät * faktor
    faktor := faktor + 1
return fakultät
```

Hier wird die Funktion *fakultät_iterativ* nur einmal aufgerufen und arbeitet dann den gegebenen Algorithmus in einer Schleife ab.

Manche Programmiersprachen (insbesondere in der Funktionalen Programmierung) erlauben keine Iteration, sodass immer die rekursive Umsetzung gewählt werden muss. Solche Sprachen setzen häufig zur Optimierung primitive Rekursionen intern als Iterationen um (insbesondere einige Interpreter für Lisp und Scheme verfahren so).

Die Rekursion ist ein wesentlicher Bestandteil einiger Entwurfsstrategien für effiziente Algorithmen, insbesondere der Teile-und-herrsche-Strategie (*Divide and Conquer*). Andere Ansätze (zum Beispiel sogenannte Greedy-Algorithmen) verlangen ein iteratives Vorgehen.

Rekursion und primitiv-rekursive Funktionen spielen eine große Rolle in der theoretischen Informatik, insbesondere in der Komplexitätstheorie und Berechenbarkeitstheorie.

Als Algorithmus ist **die Iteration oft effizienter** als der **elegantere rekursive Weg**. Implementiert wird die Rekursion in Programmen dadurch, dass sich Unterprogramme (Routinen, Subroutinen, Funktionen) selbst wieder aufrufen.

7 Prozeduren – Funktionen – Methoden

7.1 Grundsätzliche Ziele

Seit der Frühzeit des Computings werden Unterprogramme (engl. *subroutines*) eingesetzt, um verschiedene Ziele zu erreichen:

- eine bessere Strukturierung von Programmen,
- zur Abstraktion (was gemacht wird muss klar sein, aber nicht wie!)
- zur Modularisierung,
- bei mehrfacher Verwendung zum Einsparen von (Programm-)Speicherplatz, also mit dem Ziel, den Code möglichst kompakt zu halten, Ziel Wiederverwendung von Codeteilen.

Nahezu alle modernen Programmiersprachen unterstützen das Unterprogrammkonzept. In Programmiersprachen werden Unterprogramme durch **Funktionen** und **Prozeduren** repräsentiert. Gelegentlich wird *Subroutine* (oder auch einfach *Routine*) als Sammelbegriff für *Funktion* und *Prozedur* verwendet. (Aber die Sprechweisen sind in der Fachterminologie nicht einheitlich!)

Dabei wird eine Folge von Anweisungen, unter einem Namen zusammengefasst. Es können Parameter an diese Folge übergeben, und ggf. auch ein Wert zurückgeliefert werden. Die Parameter werden in der Regel durch Reihenfolge, Typ und Anzahl und/oder durch Namen festgelegt. Ein Unterprogramm wird eingesetzt, um Anweisungsfolgen, die an mehreren Stellen in einem Programmsystem verwendet werden, zusammengefasst an nur einer Stelle anzugeben.

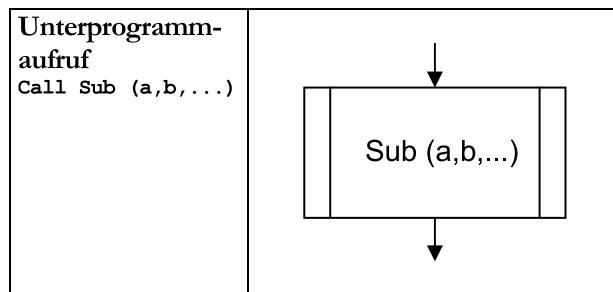
In den meisten Programmiersprachen können Unterprogramme auch weitere Unterprogramme aufrufen (Schachtelung). Kann ein Unterprogramm sich selbst aufrufen, oder können sich Unterprogramme gegenseitig aufrufen, spricht man von Rekursion, siehe oben.

In objektorientierten Sprachen wird anstelle von

- Programmaufruf eher von Nachricht und
- anstelle von Prozedur oder Funktion eher von Methode

gesprochen. Gemeint ist jedoch dasselbe Prinzip.

In Programmablaufplänen und in einem Nassi-Shneiderman-Diagramm sieht der Unterprogrammaufruf folgendermaßen aus:



7.2 Die Parameterübergabe

Unterprogramme können Argumente haben, die üblicherweise **Parameter** genannt werden. In der Unterprogrammdefinition nennt man sie **formale Parameter** (Platzhalter), denn sie werden beim Aufruf des Unterprogramms durch **aktuelle Parameter** ersetzt. Ggf. ist auch die Rückgabe von Werten über diese Parameter möglich.

Der Compiler/Interpreter vergleicht und überprüft bei der Parameterübergabe normalerweise Anzahl und Typ des aktuellen und des formalen Parametersatzes. Wenn diese nicht übereinstimmen, wird eine Fehlermeldung generiert.

Die Mechanismen zur Parameterübergabe sind allerdings sehr verschieden. Wir unterscheiden:

- Referenzparameter (*call by reference*)
- Wertparameter (*call by value*)
- Namensparameter (*call by name*) (hat nur noch historische Bedeutung!)

Referenzparameter (engl. *call by reference*) sind Parameter von Unterprogrammen, die die Übergabe und Rückgabe von Werten ermöglichen. Ihr Name kommt daher, dass der Compiler oder Interpreter die Adresse des Speicherbereichs einer Variablen oder eines Feldelements übergibt (also einen „Zeiger“ auf die Variable), die als **Referenz** aufgefasst werden kann.

Beim Aufruf des Unterprogramms wird die Adresse im formalen Parameter gespeichert. Jede Operation mit diesem formalen Parameter wirkt sofort auf den aktuellen Parameter und bleibt auch nach Verlassen des Unterprogramms erhalten, also auch im rufenden Programm.

Wertparameter (engl. *call by value*) sind Parameter von Unterprogrammen, die die Übergabe, jedoch nicht die Rückgabe von Werten ermöglichen. Beim Aufruf des Unterprogramms wird der Wert des formalen Parameters bestimmt und dieser Wert dem formalen Parameter zugewiesen. Ist der aktuelle Parameter eine einfache Variable, so entsteht eine Kopie. Änderungen an dieser Kopie (dem formalen Parameter) wirken sich im rufenden Programm nicht aus. Im Gegensatz zu einem Referenzparameter wird das rufende Programm insbesondere auch vor ungewollten Veränderungen „geschützt“. Wir haben eine echte Kapselung erreicht!

Ein **Namensparameter** (engl. *call by name*) ist ein Parameter eines Unterprogramms, der nicht bei seiner Übergabe, sondern erst bei seiner Benutzung, **entsprechend der Signatur des Parameters**, berechnet wird. Namensparameter ermöglichen sowohl die Übergabe als auch Rückgabe von Werten. Dies wurde vor allem in der Programmiersprache Cobol, daneben auch in ALGOL 60 genutzt, ist jedoch in modernen Sprachen unüblich. Ganz offensichtlich gibt es bei dieser Technik einen imminenten Nachteil: Ausdrücke werden unter Umständen mehr als einmal ausgewertet!

In einer Variante des Namensparameters, dem so genannten *call by need* wird dieses Problem behoben, indem man sich bereits ausgewertete Ausdrücke merkt. Es wird von Programmiersprachen wie Haskell verwendet. Die bedarfsgesteuerte Auswertung erlaubt zyklische Ausdrücke zu schreiben. Mit anderen Worten erhält man somit „unendliche“ Datenstrukturen. Eine besonders wichtige derartige Datenstruktur ist der „Strom“, eine potenziell unendliche und bei Bedarf berechnete Liste. Dieses wird in GPR 2 beim funktionalen Programmieren noch ausführlich behandelt werden.

Wir beschränken uns im Weiteren auf die Behandlung von *call by reference* und *call by value*. Wir wollen deren Eigenschaften gegenüberstellen.

	Referenzparameter	Wertparameter
Formale Parameter	Einfache Variablen und strukturierte Variablen	Einfache Variablen und strukturierte Variablen
Aktuelle Parameter	Nur Variablen, Felder, Feldelemente, Strukturelemente. Keine Konstanten und Ausdrücke	Beliebige Ausdrücke wie 1.0 , $2*X$, $\sin(x)$, $y[i]$
Übergabe	Als <i>Adresse</i> übergeben (geringer Aufwand bei Feldern)	Als <i>Kopie</i> (hoher Aufwand bei großen Datenstrukturen)
Zuweisung innerhalb des Unterprogramms	möglich	möglich oder verboten (je nach Programmiersprache)
Rückgabe des Wertes bei Unterprogrammende	ja	nein

Anmerkung: Moderne (optimierende) Compiler können bei Übergabe von Wertparametern ermitteln, ob eine Kopie nötig ist oder gegebenenfalls darauf verzichtet werden kann.

Zur Einordnung: In Fortran z.B. gibt es ausschließlich Referenzparameter, in C nur Wertparameter, in C++ und Java gibt es beides. Namensparameter existieren in keiner dieser Sprachen.

Frage: Wenn C nur Wertparameter unterstützt, wie können dann im Unterprogramm errechnete Werte zurückgegeben werden? C bedient sich hier eines Tricks über einen sogenannten speziellen Typ „Zeiger“ (*pointer*): Zeiger verweisen auf Variablen und hiermit kann ein ähnliches Verhalten wie bei Referenzparameter realisiert werden.

7.3 Funktionen versus Prozeduren

Einen weiteren kleinen Unterschied gibt es zu beachten: Wir unterscheiden prinzipiell zwischen Funktionen und Prozeduren:

- **Funktionen** erzeugen (errechnen) einen Wert, der an das rufende Programm als Wert der Funktion zurückgegeben wird, wie in der Mathematik üblich, z.B. $\sin(x)$. Damit kann eine Funktion u.a. in Ausdrücken als Entität auftreten: $a = 1 - \sin(x)$. Funktionen werden typischerweise in Bibliotheken (Modulen) thematisch gebündelt.
- **Prozeduren** führen eine Aktion aus. Hierdurch können entweder interne Variablen der Prozedur verändert werden (eine Zustandsänderung erwirkt werden) oder über Referenzparameter Veränderungen an Variablen im rufenden Programm bewirkt werden.

Funktionen und Prozeduren können, je nach Art der Parameterübergabe, auch Parameterwerte verändern, z.B. wenn $inc(x)$ das Argument um eins erhöht (Inkrement). Funktionen, deren Aufruf ihre Argumente und ihre Umgebung niemals verändert, heißen *nebeneffektfrei*.

7.4 Wirkung und Nebenwirkung

Gerade im Zusammenhang mit Unterprogrammen, aber nicht nur hier, ist die Diskussion des Begriffs Wirkung bedeutsam:

Wirkung bezeichnet in der Informatik die *Veränderung eines Zustands*, in dem sich ein Computersystem (oder ein anderer Teil der Welt) befindet. Beispiele sind das Verändern von Inhalten des Speichers oder die Ausgabe eines Textes auf Bildschirm oder Drucker.

Neben *Wirkung* werden synonym auch die Bezeichnungen Nebenwirkung, Nebeneffekt oder Seiteneffekt verwendet. Letzteres ist eine wortwörtliche Rückübersetzung des englischen *side effect*, was wiederum eigentlich "Nebenwirkung" bedeutet. Einfach könnte man interpretieren: Wirkung ist das erwünschte (das positive). Nebenwirkung, Nebeneffekt oder Seiteneffekt ist das unerwünschte (das negative). Leider ist der Sprachgebrauch nicht einheitlich.

Wirkungen haben in Programmiersprachen eine wichtige Funktion. Beispielsweise basieren Zuweisungsausdrücke darauf, dass es eine Wirkung gibt. Es folgt ein Beispiel, das in einer der Programmiersprachen Java, C++, C oder Python geschrieben sein könnte:

```
a = 2
```

Die Wirkung des Ausdrucks `a = 2` besteht darin, dass die Variable `a` nach Abarbeitung des Ausdrucks ihren Zustand geändert hat, indem sie nämlich den neuen "Inhalt" 2 gespeichert hat.

Bei der Betrachtung des folgenden Ausdrucks

```
(++i) - (++i)
```

könnte man auf den ersten Blick meinen, dass der Ausdruck den Wert von $(i+1)-(i+1)$, also 0, hat. Das ist **nur in Python so**, sonst aber **nicht der Fall**, da der Teilausdruck `++i` nicht nur `i+1` zurückliefert, sondern als Nebeneffekt `i` um eins erhöht und dieses Ergebnis dann zurückliefert.

Dabei gibt es einen **wichtigen Unterschied zwischen verschiedenen Programmiersprachen**: Während nämlich in Java der rechte Operand der Subtraktion einen um 1 höheren Wert als der linke hat, und dadurch das Resultat eindeutig ist (-1), gibt es **in C und C++ keine definierte Reihenfolge** für die Abarbeitung der beiden Teilausdrücke `(++i)`.

Manchmal wird sogar die Auffassung vertreten, die Notwendigkeit zur Berücksichtigung von Wirkungen erschwere generell das Verständnis von Programmen. So kommt man u.a. zu den reinen funktionalen Programmiersprachen, bei denen die Auswertung von Ausdrücken grundsätzlich keine Wirkung hat.

Ein Grundsatz: Programmieren muss so gestaltet sein, dass es für Programmierer übersichtlich und die Wirkungen möglichst eindeutig und einfach durchschaubar sind. Das heißt insbesondere, dass möglichst nur Wirkungen in einem „lokalen“ gut zu überschauenden Bereich stattfinden sollen. Insbesondere ist der Programmierer (sei er noch so intelligent) vor unerwünschten oder schwer zu überschauenden Nebeneffekten zu schützen.

Verschiedene Programmiersprachen erbringen diese Leistung auf sehr unterschiedliche Art und Weise. Das Konzept des Unterprogramms reflektiert sehr stark das jeweilige Programmierparadigma. Außerdem wählen verschiedene Sprachen aus der Fülle der Möglichkeiten unterschiedlich aus: Man kann durchaus sagen: Das Unterprogrammkonzept ist jeweils kennzeichnend für eine bestimmte Programmiersprache und verdient immer besondere Aufmerksamkeit!

8 Funktionen in Python

Python erlaubt es, Funktionen sehr einfach zu definieren, bedient sich aber auch einer Reihe von Ideen aus der funktionalen Programmierung, um einige Aufgaben zu erleichtern.

8.1 def

Funktionen, Prozeduren und Methoden werden mit der **def-Anweisung** definiert.

```
>>> def add(x,y):
    return x+y
>>>
```

Achtung: Der Funktionsrumpf muss eingerückt (*indent*) werden (macht IDLE automatisch); das Ende der Funktionsdefinition wird durch Rücknehmen der Einrückung (*dedent*) angegeben (macht IDLE bei Eingabe einer Leerzeile auch automatisch)

Eine **Funktion** wird **aufgerufen**, indem ihrem Namen ein Tupel von Argumenten unmittelbar nachgestellt wird, wie in

```
>>> add (3,4)
7
```

Die Reihenfolge und Anzahl von Argumenten müssen mit jenen der Funktionsdefinition übereinstimmen. Andernfalls wird eine **TypeError**-Ausnahme ausgelöst. Durch die Zuweisung von Werten in der Funktionsdefinition können für die Parameter einer Funktion Standardwerte voreingestellt werden, z.B.:

```
def foo(x, y, z = 42):
```

Definiert eine Funktion einen Parameter mit einer Voreinstellung (engl. *default parameter*), so ist dieser Parameter wie auch alle nachfolgende optional. Falls nicht allen optionalen Parametern in der Funktionsdefinition eine Voreinstellung gegeben wird, so wird eine **SyntaxError**-Ausnahme ausgelöst.

Voreinstellungswerte (Argumentvoreinstellungswerte) sind immer jene Objekte, die als Wert bei der Funktionsdefinition angegeben worden sind.

Voreinstellungswerte werden **nur** zum Zeitpunkt der Funktionsdefinition ausgewertet. D.h. das der Ausdruck genau einmal berechnet wird und dann als ‚pre-computed‘ Wert für jeden Aufruf gültig ist, sofern er nicht durch einen aktuellen Parameterwert ersetzt wird.

```
>>> a=10
>>> def foo(x = a):
    print x
    a = 5

>>> foo()
10
>>> foo()
10
>>> foo(3)
```

Achtung: Die Verwendung von veränderlichen Objekten (solche haben wir bisher noch nicht kennen gelernt, z.B. einer Liste) als Voreinstellungswert kann zu einem unerwarteten Verhalten führen. Beispiel: [10] ist eine Liste mit genau einem Element, nämlich 10; x.append(y) ist eine Funktion, die das Listenelement y an x hinten anfügt, also die Liste verlängert.

```
>>> a = [10]
>>> def foo(x = a):
    print x
    a.append(20)

>>> foo()
[10]
>>> foo()
[10, 20]
>>> foo()
[10, 20]
```

Was passiert in diesem Beispiel? Dem Parameter x wird die Variable a einmal bei der Ausführung der Funktionsdefinition zugewiesen, aber als Referenz, nicht als Wert! Die Funktion a.append(20) verändert bei jeder Ausführung die Funktionsexterne Liste a, so dass beim folgenden Aufruf dieser „Default-Parameter“., „**This is generally not what was intended.**“ Es verletzt stark das Kriterium der Kapselung, also **VORSICHT!**

Eine Funktion kann eine variable Anzahl von Parametern annehmen, wenn dem letzten Parameternamen ein Stern (*) vorangestellt wird:

```
>>> def printall(a,*spam):
    print(a,spam)

>>> printall(0,42,"hello world",3.45)
(0, (42, 'hello world', 3.4500000000000002))
```

In diesem Fall werden alle verbleibenden Argumente als Tupel mit der Variablen args übergeben.

Man kann Funktionsargumente auch übergeben, indem jeder Parameter explizit mit einem Namen und Wert versehen wird, wie folgt:

```
>>> def egg(w,x,y,z):
    print w,x,y,z

>>> egg(w=3,y=22,w='hello',z='world')
SyntaxError: duplicate keyword argument
>>> egg(x=3,y=22,w='hello',z='world')
hello 3 22 world
```

Bei diesen Schlüsselwort-Argumenten spielt die Reihenfolge keine Rolle. Allerdings muss man alle Funktionsparameter namentlich angeben, wenn man keine Voreinstellungswerte benutzt.

Wenn man einen der benötigten Parameter weglässt oder wenn der Name eines Schlüsselwortes mit keinem Parameternamen der Funktionsdefinition übereinstimmt, wird eine `TypeError`-Ausnahme ausgelöst. Doppelt zugewiesene Namen werden als `SyntaxError` erkannt. Wenn Positions- und Schlüsselwort-Argumente im gleichen Funktionsaufruf vorkommen, werden Positionsargumente vor allen Schlüsselwort-Argumenten zugewiesen. Beispiel:

```
>>> egg(3, 22, z='world', y='hello')
3 22 hello world
```

Wenn dem letzten Argument einer Funktionsdefinition zwei Sterne vorausgehen (`**`), werden alle weiteren Schlüsselwort-Argumente (jene, die mit keinen Parameternamen übereinstimmen) in einem Dictionary an die Funktion übergeben. Aber dies wäre ein deutlicher Vorgriff auf Verbunddatentypen. Wir kommen später auf diese Parameter-Variante zurück.

Betrachten wir jetzt die **Parameter-Übergabe und Rückgabewerte** noch etwas genauer:

Wenn eine Funktion aufgerufen wird, werden ihre **Parameter als Referenzen** (*call by reference*) übergeben. Falls ein veränderliches Objekt (z.B. eine Liste) an eine Funktion übergeben wird und in der Funktion verändert wird, so werden diese Veränderungen in der aufrufenden Umgebung sichtbar. Beispiel:

```
>>> a = [1, 2, 3, 4, 5] # a ist wieder eine Liste, also veränderlich
>>> def foo(x):
    x[3] = -55 # Verändere ein Element von x

>>> foo(a)
>>> a
[1, 2, 3, -55, 5]
```

Für unveränderliche Objekte hat dies allerdings keine Veränderung zur Folge, sondern führt zu einem Fehler:

```
>>> a = "1, 2, 3, 4, 5" # a ist jetzt ein String, also unveränderlich
>>> foo(a)

Traceback (most recent call last):
  File "<pyshell#82>", line 1, in <toplevel>
    foo(a)
  File "<pyshell#78>", line 2, in foo
    x[3] = -55 # Verändere ein Element von x
TypeError: object does not support item assignment
>>> a[3] # gibt es natürlich, warum ist das aber '2'??? Überlegen Sie
'2'
```

Die `return`-Anweisung gibt einen Wert aus der Funktion zurück. Wird kein Wert angegeben oder wird die `return`-Anweisung weggelassen, so wird das Objekt `None` zurückgegeben. Um mehrere Werte zurückzugeben, setzt man diese in ein Tupel:

```
>>> def factor(a):
    d = 2
    while (d < (a/2)):
        if ((a/d)*d == a):
            return ((a/d), d)
        d = d + 1
```

```

        return (a, 1)

>>> factor(8)
(4, 2)
>>> factor(9)
(3, 3)
>>> factor(65)
(13, 5)
>>> factor(64)
(32, 2)

```

Mehrere Rückgabewerte in einem Tupel können individuellen Variablen wie folgt zugewiesen werden:

```

>>> x, y = factor(1243) # Gibt Werte in x und y zurück
>>> x
113
>>> y-1
10

```

Python unterscheidet also nicht explizit zwischen Prozeduren und Funktionen: Wenn bei der Ausführung ein `return ()` erreicht wird, wird der in Klammern angegebene Ausdruck ausgewertet und als Funktionswert zurückgegeben. Eine Python-Funktion kann sogar beide Ausprägungen gleichzeitig annehmen.

8.2 Namensräume

Grundsätzlich soll ein Unterprogramm (eine Funktion in Python) auch der Kapselung, mindestens einer Namenskapselung, dienen.:

Jedes Mal, wenn eine Funktion ausgeführt wird, wird deshalb ein **neuer lokaler Namensraum** erzeugt. Dieser Namensraum enthält die Namen der Funktionsparameter sowie die Namen von Variablen, denen im Rumpf der Funktion zugewiesen wird.

Bei der Auflösung von Namen (z.B. wenn sie in einem Ausdruck stehen) sucht der Interpreter zunächst im lokalen Namensbereich, dann in den lokalen Bereichen aller lexikalisch umgebenen Funktionen (von innen nach außen, sofern vorhanden). Wenn nichts Passendes gefunden wird, geht die Suche im globalen Namensraum weiter. Der globale Namensbereich einer Funktion besteht immer aus dem Modul, in welchem die Funktion definiert wurde. Wenn der Interpreter auch hier keine Übereinstimmung findet, führt er eine letzte Suche im eingebauten Namensraum durch. Wenn auch diese fehlschlägt, wird eine **NameError**-Ausnahme ausgelöst.

Eine Besonderheit von Namensräumen ist die Manipulation von sogenannten **globalen Variablen** innerhalb einer Funktion. Man betrachte z.B. folgendes Codestück:

```

>>> a = 42
>>> def foo():
    a = 13

>>> foo()
>>> a
42

```

Wenn dieser Code ausgeführt wird, wird der Wert **42** ausgegeben, obwohl es den Anschein hat, dass wir die Variable **a** innerhalb der Funktion **foo()** verändern. Wenn an Variablen in einer Funktion zugewiesen wird, sind diese immer an den lokalen Namensraum der Funktion gebunden. Daher bezieht sich die Variable **a** im Funktionsrumpf auf ein gänzlich neues Objekt mit dem Wert **13**. Um dieses Verhalten zu ändern, verwendet man die **global**-Anweisung. **global** markiert lediglich eine Liste von Namen als solche, die zum globalen Namensraum gehören, und wird nur dann gebraucht, wenn globale Variablen verändert werden. Diese Anweisung darf (auch mehrfach) überall im Rumpf einer Funktion vorkommen. Allerdings muss sie nur vor der Zuweisung zu dieser Variablen erfolgen, sonst gibt es einen **SyntaxError**: `name 'x' is assigned to before global declaration`. Übrigens eine **global**-Anweisung im rufenden Programm hat auf die Funktion keine Wirkung, die Variable bliebe lokal! Ein funktionierendes Beispiel:

```
a = 42
>>> def foo():
    global a
    a=13

>>> foo ()
>>> a
13
```

Globale Variablen sind manchmal praktisch, können aber zu schwer durchschaubaren Fehlern und einer Fehlerverschleppung führen, also Vorsicht! (Für die Zukunft: Man kann eine globale Variable vermeiden, indem man stattdessen Attribute eines globalen Objekts setzt.)

Natürlich erlaubt Python auch Rekursionen: Wir nehmen als Beispiel die oben schon behandelte Fakultätsfunktion:

```
>>> def fakultaet_rekursiv(n):
    if n<=1:
        return (1)
    else:
        return (n*fakultaet_rekursiv(n-1))

>>> fakultaet_rekursiv(3)
6
>>> fakultaet_rekursiv(5)
120
>>> fakultaet_rekursiv(100)
93326215443944152681699238856266700490715968264381621468592963895217599
99322991560894146397615651828625369792082722375825118521091686400000000L
00000000000000000000
>>> float(_)
9.3326215443944151e+157
```

Sie sehen an diesem Beispiel einerseits, wie schnell die Fakultätsfunktion wächst, andererseits wie mühelos Python eine Integer in eine Long Integer wandelt, immerhin in diesem Fall eine Zahl mit 158 Dezimalstellen.

Python erlaubt die Verwendung von verschachtelten Funktionsdefinitionen. Damit entstehen auch verschachtelte Geltungsbereiche. Seit Version 2.2 wird ein Name **x**

- erst im aktuellen lokalen Geltungsbereich gesucht

- dann im Geltungsbereich aller umgebenden Funktionen (von innen nach außen)
- dann im aktuellen globalen Bereich (im Modul)
- und dann im eingebauten Bereich (`__builtin__` des Moduls)

Bei globalen Variablen beginnt die Suche im globalen Geltungsbereich. Ein Beispiel:

```
>>> def bar():
    x = 3
    def spam(): # Verschachtelte Funktionsdefinition
        print 'x is ', x # Sucht nach x im globalen Geltungsbereich
    while x > 0:
        spam()
        x = x - 1

>>> bar ()
x is 3
x is 2
x is 1
```

Wenn die verschachtelte Funktion `spam()` ausgeführt wird, ist dessen globaler Namensraum genau der Gleiche wie der für `bar()` (das Modul, in dem die Funktion definiert ist). Aus diesem Grund löst `spam()` die Symbole des Namensraumes von `bar()` auf.

In der Praxis werden verschachtelte Funktionen nur in besonderen Umständen verwendet, etwa dann, wenn ein Programm eine Funktion abhängig vom Ergebnis einer Bedingung anders definieren möchte.