

## Modul: B-PRG1: Grundlagen der Programmierung 1 und Einführung in die Programmierung EPR

### V06 Kontrollstrukturen – Prozeduren und Funktionen - Rekursion

Prof. Dr. Detlef Krömker  
Professur für Graphische Datenverarbeitung  
Institut für Informatik  
Fachbereich Informatik und Mathematik (12)

## Rückblick

Erste Kontrollstrukturen (Verzweigungen + Schleifen)

- |                    |                |
|--------------------|----------------|
| ▸ If – elif – else | Verzweigung    |
| ▸ while ...        | Schleife Typ 1 |
| ▸ for x in ...     | Schleife Typ 2 |

## Die inhaltliche Grobstruktur von PRG 1 und EPR Vorlesungen Teil 1

Woche	Montag 12 c.t.-14 Uhr – Hörsaal VI	Freitag 9.30-11 Uhr – Hörsaal V
16.10.	V 00 Begrüßung und Einführung (heute, diese Vorlesung)	V 01 Computer – Algorithmus – Programm
23.10.	V 02 Programmieren – Erste Schritte	V 03 Kontrollstrukturen 1 (Verzweigungen + Schleifen)
30.10.	V 04 Elementare Datentypen + Operatoren 1 (int, bool, none)	V 05 Elementare Datentypen + Operatoren 2 (String)
01.11.	<b>EPR 3: Erste größere Programmierarbeit im 2er-Team</b>	
06.11.	<b>V 06 Kontrollstrukturen 2 (Funktionen)</b>	V 07 Elementare Datentypen + Operatoren 3 (Float)
13.11.	V 08 Aggregierte Datentypen in Python (Builtins)	V 09 Allererste Schritte im Software-Engineering (Module)
15.11.	<b>EPR 4: Zweite größere Programmierarbeit im 2er-Team (Fahrstuhl)</b>	
20.11.	V 10 Software-Tests	V 11 Iteration und Rekursion
22.11.	PS2 Erste Aufgabe: ... Testen	

3

Vorlesung PRG 1 – V4  
Kontrollstrukturen

Prof. Dr. Detlef Krömker

## Die inhaltliche Grobstruktur von PRG 1 und EPR Vorlesungen Teil 2

Woche	Montag 12 c.t.-14 Uhr – Hörsaal VI	Freitag 9.30-11 Uhr – Hörsaal V
27.11.	V 12 OO-Programmierung – Erste Schritte	V 13 Uis systematisch entwickeln
29.11.	<b>EPR 5: Dritte größere Programmierarbeit im 2er-Team (GUI)</b>	
04.12.	V 14 GUIs programmieren 1	V 15 GUIs programmieren 2
11. 12	V 16 OO-Entwurf und –Design 1	V 17 OO-Entwurf und –Design 2
13.12.	<b>EPR 7: Vierte größere Programmierarbeit im 2er-Team (Partner*innenechsel!)</b>	
18.12.	V 18 Beispiele zu (selbstprogrammierten) Objekten	schon Weihnachspause ;-)
20.12.	PS2 Zweite Aufgabe:	

4

Vorlesung PRG 1 – V4  
Kontrollstrukturen

Prof. Dr. Detlef Krömker

## Unser heutiges Lernziele

- **Das Ziel der Strukturierung mit Unterprogrammen muss verstanden werden (Teil der Strukturierten Programmierung).**
- **Die Realisierung von eigenen Prozeduren und Funktionen muss klar sein und umgesetzt werden können.**
- *Bisher haben wir nur wenige Funktionen und Prozeduren benutzt, wie `len()`, `print()`, `input()`*
- *Den Python Mechanismus zur Parameterübergabe und die diversen Varianten muss man kennen und nutzen können.*
- *Neben der Fallunterscheidung und der Iteration ist die **Rekursion** eine grundlegende mathematische und informatische Lösungsmethode, die zu erfassen und deren Realisierungen in Programmiersprachen kennen zu lernen ist.*

5

Vorlesung PRG 1 – V4  
Kontrollstrukturen

Prof. Dr. Detlef Krömker

## Übersicht

### Prozeduren – Funktionen

- Grundsätzliche Ziele
- Die Parameterübergabe
- Funktionen versus Prozeduren

### Funktionen und Prozeduren in Python

- `def`

### ▸ Namensräume

6

Vorlesung PRG 1 – V4  
Kontrollstrukturen

Prof. Dr. Detlef Krömker

## Prozeduren – Funktionen – Methoden

### Grundsätzliche Ziele

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

- **eine bessere, übersichtlichere 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.**

## Unterprogramm (Prinzip)

Dabei wird eine Folge von Anweisungen, unter einem **Namen** zusammengefasst.

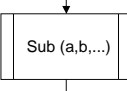
Es können **Parameter** an diese Folge übergeben, und ggf. auch ein Wert oder mehrere Werte zurückgeliefert werden.

Die Parameter werden in der Regel durch Reihenfolge, Typ und Anzahl und/oder durch Namen festgelegt.

## Graphische Repräsentationen

Sub **definieren**(x,y,...) ← **formale** Parameter

Sub **aufrufen** (a,b,...) ← **aktuelle** Parameter

Pseudocode	Ablaufplan
Call Sub (a,b,...) oder Sub (a,b,...)	

Ein **Unterprogramm** ist ein normales Programm, an das man Werte (= aktuelle Parameter) übergeben kann und das selbst Werte zurückgeben kann.

## Die Parameterübergabe

- Die Argumente, das x und y in mathematischen Funktionen  $f(x,y)$ , von Unterprogrammen nennt man **Parameter**
- In der Unterprogrammdefinition nennt man sie **formale Parameter** (Platzhalter), denn sie werden beim Aufruf des Unterprogramms durch **aktuelle Parameter** ersetzt.
- Ggf. ist auch eine 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 → Früherkennung von Fehlern..

## Mechanismen zur Parameterübergabe (klassisch)

Wir unterscheiden:

- **Wertparameter (call by value)**
- **Referenzparameter (call by reference)**
- Namensparameter (*call by name*)  
(Hat nur noch historische Bedeutung! – wurde in Algol 60 und Cobol genutzt.)
- Es gibt neuere Konzepte, wie "**call-by-Object**", was z.B. auch in Python genutzt wird – lernen wir unten kennen.

## Wertparameter (engl. call by value)

sind Parameter, die die **Übergabe**, jedoch nicht die **Rückgabe** von Werten ermöglichen.

**Beim Aufruf** des Unterprogramms **wird der Wert des aktuellen Parameters bestimmt (errechnet) und dieser Wert dem formalen Parameter zugewiesen.**

**Aufruf:** Sub (a,b,...) die aktuellen Parameter werden den formalen (x,y,...) Parametern 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.

das rufende Programm wird insbesondere vor ungewollten Veränderungen „geschützt“. Wir haben eine **echte Kapselung** erreicht!

## Referenzparameter (engl. call by reference)

sind Parameter, die die **Übergabe und Rückgabe von Werten** ermöglichen.  
 der Compiler oder Interpreter „übergibt“ eine Referenz auf den  
 Speicherbereich einer Variablen (also einen „Zeiger“ (nicht unbedingt im  
 Sinne von C,C++ auf die Variable)).

Beim Aufruf des Unterprogramms wird **die Referenz (nicht der Wert) als  
 aktueller Parameter in den formalen Parameter umgespeichert.**

Jede Operation (insbesondere auch eine Zuweisung) zu diesem formalen  
 Parameter wirkt sofort auf den aktuellen Parameter und bleibt auch nach  
 Verlassen des Unterprogramms erhalten, also **auch im rufenden  
 Programm,**

**Dieser Mechanismus kann also zur Rückgabe von Ergebnissen genutzt  
 werden.**

## Vergleich der klassischen Methoden zur Parameterübergabe

	Referenzparameter	Wertparameter
<b>Formale Parameter</b>	Einfache Variablen und strukturierte Variablen	Einfache Variablen und strukturierte Variablen
<b>Aktuelle Parameter</b>	Nur Variablen. Keine Konstanten oder Ausdrücke	Beliebige Ausdrücke wie $1.0$ , $2 * X$ , $\sin(x)$ , $y[i]$
<b>Übergabe</b>	Als <i>Referenz</i> übergeben (geringer Aufwand bei großen Datenstrukturen)	Als <i>Kopie</i> (hoher Aufwand bei großen Datenstrukturen)
<b>Zuweisung an Parameter innerhalb des Unterprogramms</b>	möglich	möglich oder verboten
<b>Rückgabe eines Wertes an den aktuellen Parameter bei Unterprogrammende</b>	ja	nein <b>Echte Kapselung!</b>

## Funktionen versus Prozeduren

- ▶ **Funktionen** erzeugen (errechnen) einen Wert, der an das rufende Programm als Wert der Funktion zurückgegeben wird, wie in der Mathematik üblich, z.B.  $\sin(30^\circ) = 0.5$ . Der Wert 0.5 ersetzt also den Ausdruck  $\sin(30^\circ)$ . 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 verändert werden, eine Zustandsänderung erwirkt werden z.B. ein `print()` veranlasst werden oder über Referenzparameter Veränderungen an Variablen im rufenden Programm bewirkt werden.

## Programmiersprachen und Unterprogrammkonzept

- ▶ Verschiedene Programmiersprachen erbringen diese Leistung auf sehr verschiedene 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 Ihre besondere Aufmerksamkeit!.

### ... jetzt zu Python



## Einige Funktionen kennen wir ja schon

- `print(...)`  
`print(value, ...)`  
 Prints the values to a stream, or to sys.stdout by default.

Funktionsname

Aktuelle Parameter  
(Argumente)

- `len(...)`  
`len(object)`  
 Return the number of items of a sequence or collection.

Dieses waren bisher alles **“built-ins” – also vordefiniert!**  
 Jetzt wollen wir unsere eigenen Funktionen “bauen”.

## Funktionen in Python – Die Definition

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

```
def spam(x, y):  
    print(x, y)  
    return x // y
```

Die Liste der  
formalen  
Parameter darf  
auch leer sein!

**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 - Aufpassen!!!)

return ist das Schlüsselwort, das veranlasst, dass der Wert ‚x+y‘ als Funktionswert dem „spam(x,y)“ zugewiesen und die Funktion beendet wird.  
return ist optional (ohne return haben wir eine Prozedur).  
return kann auch mehrfach vorkommen.

## return -Anweisung

gibt **einen Wert** aus der Funktion zurück und beendet die Ausführung des Unterprogramms.

Wird kein Wert angegeben oder wird die **return**-Anweisung weggelassen, so wird das **Objekt None** zurückgegeben.

Es können in einer Funktion mehrere **return** stehen.

Um mehrere Werte zurückzugeben, setzt man diese in ein Tupel ...  
Sehen wir später.

## Aufruf einer Funktion

Die Funktionsdefinition muss im Programmtext (lexikalisch) **vor dem Aufruf** erfolgen. (Erst dann ist der Name und Art der Funktion bekannt.)

Ihrem Namen werden die Argumente in „runden Klammern“ unmittelbar nachgestellt, wie in

```
>>> a = 1 + add(3, 4)
3 4
>>> a
8
```

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

Die **Reihenfolge und Anzahl von Argumenten (Parametern)** müssen bei **Positionsparameter** (solche haben wir hier) mit jenen der Funktionsdefinition übereinstimmen.

Anderenfalls wird eine **TypeError-Fehler** ausgelöst.

## Die aktuellen Parameter dürfen Ausdrücke (expressions) sein.

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

```
a = 8 + 2
>>> add(a, 8 // 4)
10 2
12 #dies ist der Funktionswert von
    #add(a, 8 / 4)
```

Offensichtlich werden die Werte der Ausdrücke berechnet und dann an dem formalen Parametern zugewiesen ... also **call by value**???

## Wie macht nun Python die Parameterübergabe genau (1)?

Python nutzt einen „Zwitter“! -- "Call-by-Object" (manchmal auch "Call by Object Reference" oder "Call by Sharing" genannt).

- 1) Wenn man an **unveränderliche** (*immutable*) aktuelle Parameter (wie Integers, Strings, **kurz alle Typen, die wir bisher kennen**) an eine Python-Funktion übergibt, verhält sich die Übergabe wie eine **Wertübergabe (wie bei call by value)**.

Die Referenz auf das **unveränderliche** (immutable) Objekt wird an den formalen Parameter der Funktion übergeben. Innerhalb der Funktion kann der Inhalt des Objektes nicht verändert werden (dann würde ja ein neues Objekt unter gleichem Namen entstehen).

## Verhalten bei Argumenten (aktuellen Parametern), die unmutable sind:

```
>>> def spiel(x,y):
        x = 'hallo'
        print(x,y)
        return y
```

```
>>> a = 8 + 2
>>> spiel(a, 8%3)
hallo 2
2
>>> print(a)
10
```

- Offensichtlich wurde **a** im rufenden Programm **nicht verändert**.
- ... verhält sich wie **call by value**.

## Wie macht nun Python die Parameterübergabe genau (2)?

**Python nutzt einen „Zwitter“! -- "Call-by-Object"** (manchmal auch "Call by Object Reference" oder "Call by Sharing").

2) Wenn man **veränderliche** (*mutable*) aktuelle Parameter hat, dann ...

Solche „mutable“ Variablen kennen wir noch nicht (nächste Woche), aber wir können schnell einen solchen Typ kennenlernen, z.B. eine Liste

## Listen als Beispiel für mutable Datentypen

Listen werden als **durch Komma getrennte** Werte in **eckigen Klammern** notiert: [x, y, z]:

```
>>> liste = [1, 2, 3] #eine Liste mit drei Integer-Items
[1, 2, 3]
>>> liste[1] #holt das Element 2 in der Liste
2
id(liste)
46761424
liste[1] = 5 #hier ist der entscheidende Unterschied
              # Wertzuweisung zu einem Listenelement
liste
[1, 5, 3]
id(liste) #das zweite Item wurde "in place" verändert
46761424 # es ist aber dasselbe Objekt
```

## Wie macht nun Python die Parameterübergabe genau (2)?

**Python nutzt einen „Zwitter“! -- "Call-by-Object"** (manchmal auch "Call by Object Reference" oder "Call by Sharing").

2) Wenn man als aktuelle Parameter **veränderliche** (*mutable*) Typen hat:

- Wird diesen Variablen **ein neuer Wert zugewiesen** (=), so wirkt sich das im **rufenden Programm nicht aus!** (so ist die Wirkung **lokal**, nur im Unterprogramm).
- Wird aber an den Datentypen „**nur**“ **etwas „verändert“** (durch Zugriff über den Index, durch Slicing, oder durch verändernde Funktionen (x.append(), x.insert())), **so wirkt sich das auf das rufende Programm aus.**

Das müssen wir tatsächlich noch etwas genauer betrachten.

## Verhalten bei Argumenten die *mutable* sind:

```
def spiel_2(x,y):
    print(y, x)
    print(id(y),id(x))
    y[1] = 5 # Veränderung
    print(id(y))
    y.append(7) # Veränderung
    print(id(y))
    x = [10,11,12] # Zuweisung
    print(id(x))
    print(y, x)

liste = [1, 2, 3]
liste_10 = [10,20]
print(id(liste))
print(id(liste_10))
spiel_2(liste_10,liste) # Aufruf
```

[1, 2, 3] [10, 20]  
675539059720 675499001864

675539059720 # alte id  
675539059720 # alte id  
675536356232 # neue id  
[1, 5, 3, 7] [10, 11, 12]

675539059720  
675499001864

## Verhalten bei Argumenten die *mutable* sind:

```
print(liste)
print(liste_10)
print(id(liste))
print(id(liste_10))
```

[1, 5, 3, 7] # verändert  
[10, 20] # unverändert  
675539059720  
675499001864

- x wird mit dem aktuellen Parameter `liste_10` aufgerufen.
- y wird mit dem aktuellen Parameter `liste` aufgerufen
- x wird in `spiel_2` ein neuer Wert im Rumpf zugeordnet
- y wird in `spiel_2` im Rumpf verändert.
- `liste[]` (im Rumpf y) "**exportiert**" diese Änderung ins rufende Programm (wie bei call by reference) und die id bleibt dieselbe
- `liste_10[]` (im Rumpf x) hingegen nicht (wie bei call by value).

## Also merken:

- Das Schlüsselwort **def** realisiert beides: **Funktionen und Prozeduren**. Funktionswerte werden mit dem Schlüsselwort **return** zurückgegeben.
- Python nutzt zur Parameterübergabe: **"Call-by-Object", d.h.**
  - Für **unmutable-Variablen** wirkt dies wie „call-by-value“ (eine sichere Variante im Sinne der Fehlerausbreitung!)
  - Für **mutable-Variablen** kommt es auf die Art der Veränderung an:
    - **Zuweisungen** an diese Variable wirken nur **lokal**.
    - **Veränderungen** werden **in place** ausgeführt, **also exportiert**.
- Also: Veränderungen kann man zur **Parameterrückgabe wie bei „call by reference“ nutzen**, siehe vorheriges Beispiel `liste[]`.

## Weitere Parameter-Varianten: Default-Parameter (default = vorgegeben, voreingestellt)

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

- **Default-Parametern** (Voreinstellungswerte) wird schon mit der Funktionsdefinition ein Wert zugewiesen.
- Dieser Wert kann beim Aufruf überschrieben werden. Dieser (vorbesetzte) Parameter kann beim Aufruf auch weggelassen werden (ist also optional).
- Die Default-Werte werden **nur zum Zeitpunkt der Funktionsdefinition** im definierenden Gültigkeitsbereich **ausgewertet** (wenn die Definition der Funktion vom Interpreter gelesen wird – nicht bei jedem Aufruf der Funktion). 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.

## Probieren wir es aus:

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

>>> foo(3,5)
3 5 42
>>> foo(3,5,1)
3 5 1
>>> foo(3,5)
3 5 42
```

## Hier gibt es aber eine Falle! – Aufpassen!

```
def f(a, L=[]):
    L.append(a)
    return L
print(f(1))
print(f(2))
print(f(3))
```

druckt

```
[1]
[1, 2]
[1, 2, 3]
```

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
print(f(1))
print(f(2))
print(f(3))
```

druckt

```
[1]
[2]
[3]
```



## Weitere Parameter-Varianten: Schlüsselwortargumente

```
>>> def foo(x, y, example):
        print (x,y,example)

>>> foo (1, 2, example = 3)
1 2 3
>>> foo (5, example = 42, y = 10)
5 10 42
```

- sind beim **Funktionsaufruf** in der Form „Schlüsselwort = Wert“ anzugeben.
- Schlüsselwortargumente müssen **nach** positionsabhängigen Argumenten kommen.
- Alle übergebenen Schlüsselwortargumente müssen jeweils auf eines der Argumente passen, die für die Funktion definiert sind, wobei ihre Reihenfolge aber unwichtig ist.
- Das gilt auch für nicht-optionale Argumente (beispielsweise y).
- Keinem Argument darf mehr als ein Wert zugewiesen bekommen.

## Beliebig lange Argumentlisten

(nur der Vollständigkeit halber, besprechen wir im Zusammenhang mit den Compound-Datenstrukturen)

- Ein Parameter der Form **\*name** erwartet ein **Tupel**, das alle **positionsabhängigen Argumente** enthält, die über die Anzahl der definierten Parameter hinausgeht.
- Ein Parameter der Form **\*\*name** erwartet ein Dictionary (Schlüssel:Wert Paare), **das alle Schlüsselwortargumente enthält**, bis auf die, die in der Definition schon vorkommen.
- **\*name** muss in der Parameterliste vor allen Schlüsselwortargumenten und vor **\*\*name** kommen.
- Diese Formen werden nicht sehr häufig genutzt. Es ist sehr schwierig hier die Übersicht zu behalten.

## Umbenennungen von Funktionen

Eine interessante Eigenschaft von Python-Funktionen zeigt rechts stehendes Beispiel:

Man kann durch Zuweisung Funktionen einfach umbenennen.

**Bitte nur wissen, nicht nutzen!**

```
>>> def spiel(x,y):
    x = 'hallo'
    print(x,y)

>>> spiel
<function spiel at 0x0366C930>
>>> s = spiel
>>> s(2,3)
hallo 3
>>> spiel is s
True
```

## Erst einmal etwas durchatmen: Übersicht

### Prozeduren – Funktionen

- Grundsätzliche Ziele
- Die Parameterübergabe
- Funktionen versus Prozeduren

### Funktionen und Prozeduren in Python

- def

### ▸ Namensräume

### ▸ Rekursive Grundstrukturen

## Namensräume (1)

- Grundsätzlich soll ein Unterprogramm (eine Funktion) in Python auch der Kapselung dienen, **mindestens einer Namenskapselung**.
- Jedes Mal, wenn eine Funktion oder Prozedur aufgerufen wird, wird ein **neuer lokaler Namensraum** erzeugt. Dieser Namensraum enthält die Namen der **formalen Funktionsparameter** sowie die Namen von Variablen, denen im Rumpf der Funktion Werte zugewiesen werden. Diese Bezeichner werden mit den ihnen zugeordneten Objekten in einer **lokalen Symboltabelle (=Namenstabelle)** abgelegt.
- Funktionen und Prozeduren sind wichtige Instrumente um die **Übersicht über benutzte Namen (Bezeichner)** zu behalten. Sie sind prinzipiell zur Kapselung (Vermeidung der Fehlerausbreitung) geeignet.

## Namensräume (2)

- Die aktuellen Parameter (Argumente), die beim Funktionsaufruf übergeben werden, werden den formalen Parametern der Parameterliste zugeordnet und gehören damit zur lokalen Symboltabelle der Funktion, wenn Ihnen ein Wert zugeordnet wird.
- Das heißt, Argumente werden auch im Fall des "call by value" als Referenz übergeben (wobei der Wert allerdings immer eine Referenz auf ein Objekt ist, nicht der Wert des Objektes selbst, deswegen **Call by Object Reference**)

### Namensräume (3) Namens-Auflösung

1. Bei der Auflösung von Namen (z.B. wenn sie in einem Ausdruck stehen und der Wert angefragt wird) sucht der Interpreter zunächst im **lokalen Namensbereich**,
2. **dann in den lokalen Bereichen** aller lexikalisch (d.h. im Text des Source Codes) umgebenen (=übergeordneter) Funktionen (def innerhalb eines def). von innen nach außen, sofern vorhanden).
3. 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.
4. Wenn der Interpreter auch hier keine Übereinstimmung findet, führt er eine letzte Suche **im eingebauten Namensraum (built-ins)** durch.

Wenn auch diese fehlschlägt, wird eine NameError-Ausnahme ausgelöst.

### Namensräume (4)

Darum ist es **ohne weiteres nicht möglich**, einer globalen Variablen innerhalb des lokalen Namensraums einer Funktion einen Wert zuzuweisen. (Dadurch würde stattdessen eine *neue*, namensgleiche lokale Variable erzeugt, die die namensgleiche globale Variable überdeckt und dadurch auch den lesenden Zugriff auf diese globale Variable verhindert.

**Ein lesender Zugriff** auf globale Variablen (= Variablen des rufenden Programms) **ist ansonsten immer möglich**, ein schreibender Zugriff nur unter Verwendung der **global** Anweisung.

## Globale Variablen -- VORSICHT, VORSICHT!

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

>>> foo()
>>> a
42
```

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

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

Globale Variablen (gekennzeichnet durch das Schlüsselwort `global` erscheinen manchmal praktisch, können aber zu schwer durchschaubaren Fehlern und einer Fehlerverschleppung führen, **also Vorsicht!** (Man kann eine globale Variable vermeiden, indem man stattdessen Attribute eines globalen Objekts setzt.)

## Rekursive Grundstrukturen

**Rekursion**, auch *Rekurrenz* oder *Rekursivität*, bedeutet Selbstbezüglichkeit (von lateinisch *recurrere* = zurücklaufen; engl. *recursion*).

tritt immer dann auf, wenn etwas auf sich selbst verweist.

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 selbst spricht.

## Beispiele

Eine etwas subtilere Form der Rekursion (**indirekte Rekursion**) kann auftreten, wenn zwei Dinge gegenseitig aufeinander verweisen.

„Der folgende Satz ist wahr.“ - „Der vorhergehende Satz ist nicht wahr.“

„Um Rekursion zu verstehen, muss man erst einmal Rekursion verstehen.“

„Kürzeste Definition für Rekursion: siehe Rekursion.“

## Definition (Informatik)

Als **Rekursion** bezeichnet man in der Informatik den **Aufruf** eines Unterprogramms **durch sich selbst**.

Ohne geeignete Abbruchbedingung geraten solche rückbezüglichen Aufrufe in einen so genannten infiniten Regress. (wie eine Endlosschleife)

## Grundidee der rekursiven Definition

Der Funktionswert  $f(n+1)$  einer Funktion  $f: \mathbb{N}_0 \rightarrow \mathbb{N}_0$  ergibt sich durch Verknüpfung bereits vorher berechneter Werte  $f(n)$ ,  $f(n-1)$ , ...

Falls außerdem die Funktionswerte von  $f$  für hinreichend viele Startargumente bekannt sind, kann jeder Funktionswert von  $f$  berechnet werden.

Bei einer rekursiven Definition einer Funktion  $f$  ruft sich die Funktion so oft selbst auf, bis ein vorgegebenes Argument erreicht ist, so dass die Funktion terminiert (abbricht).

**Vorgehensweise in der funktionalen Programmierung (siehe PRG 2)**

## Beispiel

Die Funktion  $sum(n)$  berechnet die Summe der ersten  $n$  Zahlen.

also:  $sum(n): \mathbb{N}_0 \rightarrow \mathbb{N}_0: sum(n) = 0 + 1 + 2 + \dots + n$

Anders ausgedrückt:  $sum(n) = sum(n-1) + n$  (**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  $sum(0) = 0$  (**Rekursionsanfang, base case**) fest.

## Beispiel (2)

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

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

Zu  
**Rekursion vs. Iteration**  
wird es noch eine weitere Vorlesung geben!

## Zusammenfassung

Wieder sehr viel Stoff: **Machen Sie die Quiz:**

**Kontrollstrukturen: Verzweigungen, Schleifen, Prozeduren  
/Funktionen und auch Strings**

*Achtung: Q3a und Q3b nur noch bis Freitag, 10.11. – 8.00 Uhr gegen  
Punkte bearbeitbar!*

Bereiten Sie sich auf die Übungen vor: **Tutoren löchern!**

Wir haben diese Konzepte eingeführt ...

*Die Praxis, das Programmieren müssen Sie jetzt üben!*



## **Ausblick ... morgen Dienstag**

**Für alle Informatiker\*innen und Bioinformatiker\*innen**

## **Einführung in das Studium: Die Sache mit dem Lernen**

**... nächsten Freitag dann wieder EPR**

**V07: Der Datentyp Float**