www.uni-frankfurt.de

Modul: Programmierung B-PRG
Grundlagen der Programmierung 1 – Teil 1 – V7

Prozeduren – Funktionen – Methoden

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

Rückblick

Wir haben

Verzweigung und Iteration

als grundlegende mathematische und informatische Lösungsmethoden kennen gelernt.

Unser heutiges Lernziel

Rekursion *ist als weitere grundlegende mathematische und informatische Lösungsmethode zu erfassen und deren Realisierungen in Programmiersprachen kennen zu lernen, insbesondere* ***Unterprogrammtechniken****.*

Die Mechanismen zur Parameterübergabe und die Konzepte Prozedur, Funktion, Methode müssen unterschieden werden.

Unterprogramme dienen auch zur Strukturierung von Programmen, mit den Aspekten Namensräume, Kapselung, Abstraktion.

Und schließlich: die konkrete Ausprägung in Python.

Übersicht

Rekursive Grundstrukturen

Prozeduren – Funktionen – Methoden
- Grundsätzliche Ziele
- Die Parameterübergabe
- Funktionen versus Prozeduren
- Wirkung und Nebenwirkung

Funktionen in Python
- def
- Namensräume

Funktionen in Modulen, Paketen und Klassen

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 selber spricht.

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

Sprachliche Beispiele

„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 und Mathematik)

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 später) durch sich selbst.

Ohne geeignete Abbruchbedingung geraten solche rückbezüglichen Aufrufe in einen so genannten infiniten Regress (umgangssprachlich auch Endlosschleife genannt).

Grundidee der rekursiven Definition

Der Funktionswert $f(n+1)$ einer Funktion f: $\mathbf{N_0} \rightarrow \mathbf{N_0}$ ergibt sich durch Verknüpfung bereits vorher berechneter Werte $f(n)$, $f(n-1)$, ...

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 selber auf, bis ein vorgegebenes Argument (meistens 0) 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): $\mathbf{N_0} \rightarrow \mathbf{N_0}$: *sum*(n) = 0 + 1 + 2 +...+ 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**) fest.

Beispiel (2)

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

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

Beispiel: Fakultät einer Zahl

Für alle natürlichen Zahlen

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

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)

Realisierungen

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

Primitive Rekursion

stets durch eine Iteration ersetzbar

Kennzeichen:
der Aufruf-Baum enthält keine Verzweigungen, das heißt er ist eigentlich eine Aufruf-Kette:

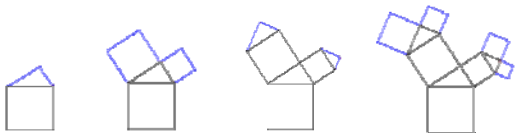
das ist immer dann der Fall, wenn eine rekursive Funktion sich selbst jeweils nur **einmal** aufruft,

insbesondere am Anfang (***Head Recursion***) oder nur am Ende (***Tail Recursion***) der Funktion.

Pythagoräischer Baum

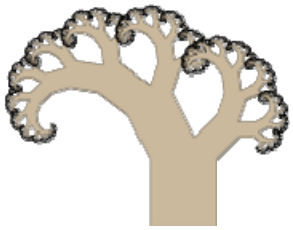
Algorithmus:

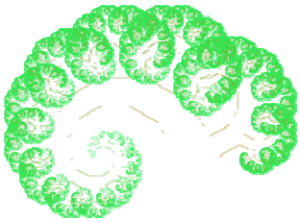
- Errichte über zwei gegebenen Punkten ein *Quadrat*
- Auf der Oberseite zeichne ein *Dreieck* mit definierten Winkeln bzw. Höhe
- Rufe diese Funktion für die beiden Schenkel dieses Dreieckes auf



Ergebnisse

Nach vielen Rekursions-Schritten und entsprechenden Einfärbungen ähnelt dieses rekursiv definierte Gebilde dann immer mehr einem Baum:





Zusammenfassung Rekursion

Rekursion bedeutet Selbstbezüglichkeit

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

muss nicht immer **direkt** auf sich selbst verweisen (**direkte Rekursion**).

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.

Prozeduren – Funktionen – Methoden
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.

Unterprogramm (Prinzip)

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 aufzuführen.

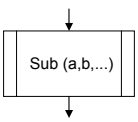
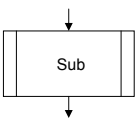
Sprechweisen in objektorientierten Sprachen

Anstelle von

- Programmaufruf sagt man **Nachricht** und
- anstelle von Prozedur oder Funktion eher **Methode**

Gemeint ist jedoch dasselbe Prinzip. .

Graphische Repräsentationen

Pseudocode	Ablaufplan	Nassi-Shneiderman
Call Sub (a,b,...) oder Sub (a,b,...)		

Achtung: Bei Nassi-Shneiderman ist eigentlich **kein** Symbol definiert, man verwendet ggf. das entsprechende Symbol aus Programmablaufplänen.

Die Parameterübergabe

- Die Argumente 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 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.

Mechanismen zur Parameterübergabe

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, die die Übergabe und Rückgabe von Werten ermöglichen.

der Compiler oder Interpreter „übergibt“ die Adresse des Speicherbereichs einer Variablen (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 (insbesondere auch eine Zuweisung) 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, die die Übergabe, jedoch nicht die Rückgabe von Werten ermöglichen.
Beim Aufruf des Unterprogramms wird der Wert des aktuellen 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.

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

Ein Namensparameter (englisch *call by name*)

ist ein Parameter, der nicht bei seiner Übergabe, sondern erst bei seiner Benutzung, **entsprechend der Signatur des Aktualparameters** berechnet wird.

Namensparameter ermöglichen sowohl die Übergabe als auch Rückgabe von Werten.

In Cobol und ALGOL 60 genutzt, ist jedoch in modernen Sprachen unüblich.

imminenter Nachteil: Ausdrücke werden unter Umständen mehr als einmal ausgewertet!

Variante des Namensparameters: *call by need*

dieses Problem wird behoben, indem man sich bereits ausgewertete Ausdrücke merkt (z.B. in Haskell)

Dieses wird in GPR 2 beim funktionalen Programmieren noch ausführlich behandelt werden.

Vergleich

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

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 (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 verändert werden (eine Zustandsänderung erwirkt werden) oder über Referenzparameter Veränderungen an Variablen im rufenden Programm bewirkt werden.

Wirkung und Nebenwirkung

Wirkung bezeichnet in der Informatik die ***Veränderung eines Zustands***, in dem sich ein Computersystem 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. eine wortwörtliche Rückübersetzung des englischen ***side effect***, was wiederum eigentlich "Nebenwirkung" bedeutet.

Also Wirkung ist das erwünschte (das positive). Nebenwirkung ist das unerwünschte (das negative).

Leider ist der Sprachgebrauch nicht einheitlich.

Auffassungen und Grundsätze

Auffassung: die Notwendigkeit zur Berücksichtigung von Wirkungen erschwert generell das Verständnis von Programmen. ➔ funktionalen Programmiersprachen, bei denen die Auswertung von Ausdrücken grundsätzlich keine Wirkung hat.

Ein Grundsatz: Programmieren muss so gestaltet sein, das 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.

Programmiersprachen und Unterprogrammkonzept

- 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!

Funktionen in Python – Die Definition

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)

Aufruf einer Funktion

ihrem Namen wird ein Tupel von Argumenten unmittelbar nachgestellt, wie in

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

Die Reihenfolge und Anzahl von Argumenten (Parametern) müssen mit jenen der Funktionsdefinition übereinstimmen. Anderenfalls wird eine TypeError-Ausnahme ausgelöst.

Default-Parameter (*default* = vorgegeben)

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

- Voreinstellungswerte sind 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.

Beispiel 1: Parameter *unmutable*

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

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


Beispiel 2 Parameter *mutable*

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

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

Variante 1:

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 übergeben.

Variante 2

Man kann Funktionsargumente auch übergeben, indem jeder Parameter
explizit mit einem Namen und Wert versehen wird.
Bei Schlüsselwort-Argumenten spielt die Reihenfolge keine Rolle.
Allerdings muss man alle Funktionsparameter namentlich angeben,
wenn man keine Voreinstellungswerte benutzt.

```
>>> 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
```

Parameter-Übergabe und Rückgabewerte

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.

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

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


Zuweisung an unveränderliche Objekte

führt zu einem Fehler:

```
>>> a = "1, 2, 3, 4, 5" # a ist 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'???
'2'
```

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)
```

Namensräume (1)

- Grundsätzlich soll ein Unterprogramm (eine Funktion in Python) auch der Kapselung, mindestens eine 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).

Namensräume (2)

- 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.

Globale Variablen

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

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

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

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

Globale Varaiblen sind 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.)

Funktionen in Modulen, Paketen und Klassen

Dieses Kapitel sollten Sie im Skript nachlesen!

Zusammenfassung

Rekursion *ist eine grundlegende mathematische und informatische Lösungsmethode*

Unterprogramme: Prozedur, Funktion, Methode

Mechanismen zur Parameterübergabe

Unterprogramme dienen auch zur Strukturierung von Programmen, mit den Aspekten Namensräume, Kapselung, Abstraktion.

konkrete Ausprägung in Python

Fragen und (hoffentlich) Antworten

Ausblick

Strukturierte Datentypen:

- Listen
- Tupel
- Mengen
- Dictionaries

- ... Bäume, usw.

Danke für Ihre Aufmerksamkeit