Programmierkurs Python I

Michaela Regneri

2009-11-12

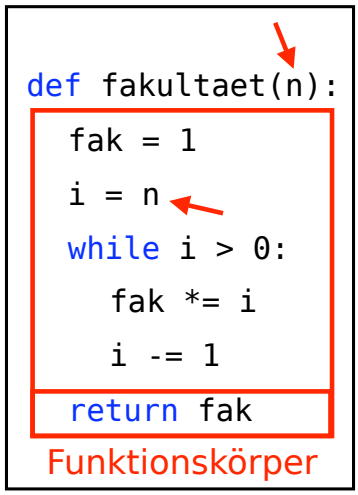
(Folien basieren auf dem gemeinsamen Kurs mit Stefan Thater)

Übersicht

- Funktionen
- Rekursion
- Sammeltypen:
  - Listen, Tuples
  - Mengen
  - Dictionaries
- `for`-Schleifen

Funktionen

- Funktionen sind (hier) zusammengehörende Code-Blöcke, die wiederverwendbar sind
- wird eine Funktion aufgerufen, wird all ihr Code ausgeführt
- Funktionen haben Parameter, auf die die Funktion zugreifen kann
- Funktionen können Rückgabewerte haben; `x = fun()` heißt, `x` bekommt den Wert, den `fun()` mit `return` zurück gibt




```
>>> fakultaet(4)
24
```


Funktionen - Syntax

- Funktionsdefinitionen beginnen mit `def`
- beliebig viele (evtl. 0) Parameter, werden mit Kommata getrennt
- Rückgabewert mit `return` markiert; Funktionen ohne Rückgabewert haben keine `return`-Zeile

```
def fun(n,m,k):
    ....
    return ret
```

Funktionen - Aufrufe

- Aufruf mit Parameter p1-n:
  `funktionsname(p1,p2,...)`
- Funktionsaufrufe sind Ausdrücke, die zum Rückgabewert der Funktion auswerten
- im Funktionsablauf selbst werden die Parameter-Variablen mit den Werten aus dem Aufruf (in der angegebenen Reihenfolge) instantiiert:

```
def fun(n,m,k):
    print('var',n,m,k)
    return m
```

```
> fun(1,2,3)
var 1 2 3
2
```

Rückgabewert

Funktionen - Variablen

- Funktionen können u.a. auf *lokale* Variablen zugreifen
  - die Parameter
  - in der Funktion neu definierte Variablen
- außerhalb der Funktion sind die lokalen Variablen nicht sichtbar
- *Manipulation* von Variablen innerhalb einer Methode funktionieren nur mit lokalen Variablen

```
def fakultaet(n):
    fak = 1
    i = n
    while(i > 0):
        fak *= i
        i -= 1
    return fak
```

```
counter = 0
def countup():
    counter += 1
```

kompiliert nicht

Rekursion

- Funktionen können andere Funktionen aufrufen
- im Speziellen können Funktionen auch sich selbst aufrufen, das nennt man *Rekursion*
- Rekursion ist ein mächtiges Werkzeug, mit dem man viele Algorithmen elegant ausdrücken kann
- Vorsicht: Wie auch bei Schleifen muss man darauf achten, dass die Rekursion irgendwo endet!

Rekursion - Fakultätsfunktion

- die Fakultäts-Funktion rekursiv:

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

Rekursionsabbruch für 0 und 1

Rekursion mit absteigendem n

Rekursion - Fibonacci

- Die Fibonacci-Zahlen ist eine Zahlenfolge, die rekursiv für natürliche Zahlen definiert ist:
  fibonacci(0) = 0
  fibonacci(1) = 1
  fibonacci(n) = fib(n-1) + fib(n-2)

```
def fibonacci(n):
  if n <= 1:
    return n
  else:
    return fibonacci(n-1) + fibonacci(n-2)
```


Sammeltypen

- Sammeltypen sind eingebaute Datenstrukturen, die verschiedene Elemente kombinieren können
  - Listen: Sammlung von Elementen, feste Reihenfolge
  - Mengen: ungeordnete Sammlung von Elementen
  - Wörterbücher: Abbildungen von Schlüssel auf Werte
- für Objekte `s` von irgendeinem Sammeltyp:
  - `len(s)`: Anzahl der Elemente in `s`
  - `s.clear()`: entfernt alle Elemente aus `s`
  - `s1 == s2`: (Wert-)Gleichheit von `s1` und `s2`

Listen: `list`

- eine Liste ist eine geordnete Sammlung von Werten
- man kann sie als literal aufschreiben:
  `liste = [‘a‘, ‘Hallo‘, 1, 3.0, [1,2,3]]`
- die Listenelemente müssen nicht den gleichen Typ haben (s.o.)
- Zugriff auf Listenelemente mit Indizes :

```
> liste[0]
‘a‘
```

```
> liste[-1]
[1,2,3]
```

```
> liste[-1][1]
2
```

```
> liste[5]
IndexError: list index out of range
```


Listen - Methoden und Operatoren (1)

- Elemente hinzufügen:
  - Element anhängen: `list.append(elem)`
  - Element an Stelle i einfügen: `list.insert(elem,i)`
- Listen konkatenieren:
  - entweder: `newlist = list1 + list2`
  - oder: `list1.extend(list2)`
- Elemente löschen:
  - `li.remove(el)` löscht das erste `el` in der Liste `li`
  - `del li[n]` löscht das Element mit Index n
- Zugehörigkeit und Nicht-Zugehörigkeit:
  `elem in list` bzw. `elem not in list`

Listen - Methoden und Operatoren (2)

- Index des ersten Auftretens von elem in list: `list.index(elem)`
- Wie oft ist elem in list? `list.count(elem)`
- Liste invertieren: `list.reverse()`
- Liste sortieren: `list.sort()` (nur bei Typgleichheit)

```
> li = [5,2,7]
> li.reverse()
> li
[7, 2, 5]
```

```
> li = [5,2,7]
> li.sort()
> li
[2, 5, 7]
```

```
> li = [[1,2],[1,2,3], [3,2], [1,3]]
> li.sort()
> li
[[1, 2], [1, 2, 3], [1, 3], [3, 2]]
```


Listen - Methoden und Operatoren (3)

- Man darf Listen auch mit ganzen Zahlen „multiplizieren“:

```
> li = [1,2,3]
> li = li * 3
> li
[1,2,3,1,2,3,1,2,3]
```

- Bei `liste*n` erscheint der Inhalt von `liste` `n` mal in der Ergebnis-Liste; `n <= 0` ergibt die leere Liste
- Achtung: es werden keine sog. *tiefen* Kopien erzeugt! (Mehr dazu später)

```
> li = [[]] * 3
> li[0].append(1)
> li
[[1],[1],[1]]
```

Listen - *Slicing* (1)

- mit dem Slicing-Operator kann man sich Teillisten einer Liste zurückgeben lassen
  - `liste[i:]` ist die Teilliste von i bis zum Ende von `liste`
  - `liste[i:j]` ist die Teilliste von `i` bis (ausschließlich) `j`
  - `liste[i:j:k]` macht dabei immer `k`-er Schritte

```
> zahlen = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
> zahlen[2:8]
[2, 3, 4, 5, 6, 7]
> zahlen[2:8:2]
[2, 4, 6]
> zahlen[8:2:-1]
[8, 7, 6, 5, 4, 3]
```


Listen - *Slicing* (2)

- mit Slicing lassen Listen sich elegant modifizieren

`del liste[0:3]` löscht die ersten 3 Elemente in `liste`

`del liste[0:5:2]` löscht jeden zweiten Eintrag ab dem 1. bis zum 5. Element in `list`

`list1[0:3] = list2` ersetzt die ersten 3 Elemente von `list1` durch die Elemente von `list2`

`list1[0:5:2] = list2` ersetzt jeden zweiten Eintrag ab dem 1. und bis zum 5. Element in `list` durch aufeinanderfolgende Einträge von `list2` (list2 muss genauso viele Elemente enthalten wie `list1[0:5:2]`!)

Listen - *Slicing* (3)

```
> zahlen = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
> zahlen[2:5] = [2,2,3,3,4,4]
> zahlen
[0, 1, 2, 2, 3, 3, 4, 4, 5, 6, 7, 8, 9, 10]
> zahlen[0:9:2] = ['a','a','a','a','a']
> zahlen
['a', 1, 'a', 2, 'a', 3, 'a', 4, 'a', 6, 7, 8, 9, 10]
```


Tupel: `tuple`

- ähnlich wie Listen: `('a',1, 'b')`, aber unveränderlich
- Intialisierung:
  - 0 Elemente: `tupel =()`
  - 1 Element elem: `tupel = elem,`
  - mehr Elemente: `tupel = elem1, elem2, elem3`
- Zugriff auf Elemente mit , sonst keine Methoden
- effizienter als Listen
- *sequence unpacking:* (funktioniert auch mit Listen)

```
> tu = 'a', 2, [2,3]
> x,y,z = tu
> z
[2,3]
```

Mengen: `set`

- Mengen sind ungeordnete Sammeltypen, die jedes Element höchstens ein mal enthalten

```
> numbers = [1, 2, 3, 1, 1]
> menge = set(numbers)
> menge
{1, 2, 3}
```

- als Literal: `menge ={1,2,4,5}` (leere Menge: `set()`)
- oder Definition über einen anderen Sammeltyp
- doppelte Elemente werden eliminiert
- effizient Werte auf (Listen-)Zugehörigkeit testen
- *in Mengen dürfen nur unveränderliche Typen enthalten sein!* (Zahlen, Strings, Booleans, ...)

Mengen - Methoden und Operatoren (1)

- `elem` hinzufügen: `set.add(elem)`
- `elem` entfernen:
  - `set.remove(elem)` (Fehler wenn `elem` nicht vorhanden)
  - `set.discard(elem)` (entfernt `elem` falls vorhanden)
- alle Elemente aus `set2` zu `set1` hinzufügen: `set1.update(set2)`
- Zugehörigkeit und Nicht-Zugehörigkeit: `elem in set` bzw. `elem not in set`

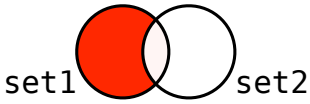
Mengen - Methoden und Operatoren (2)

Methoden können auch andere Sammeltypen als 2. Argument haben, Operatoren benötigen zwei Mengen.

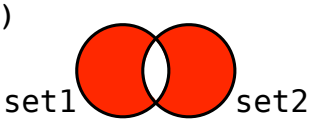
- Teilmenge / Obermenge (Rückgabe: True/False):
  - `set1.issubset(set2)` bzw. `set1.issuperset(set2)`
  - `set1 <= set2` bzw. `set1 >= set2`
- Vereinigungsmenge / Schnittmenge (Rückgabe: die neue Menge)
  - `set1.union(set2)` bzw. `set1.intersection(set2)`
  - `set1 | set2` bzw. `set1 & set2`

Mengen - Methoden und Operatoren (3)

- Differenzmenge (Rückgabe: neue Menge mit Elementen, die in `set1` aber nicht in `set2` sind)
  - `set1.difference(set2)`
  - `set1 - set2`



- Symmetrische Differenzmenge (Rückgabe: neue Menge mit Elementen, die entweder in `set1` oder in `set2`, aber nicht in beiden sind)
  - `set1.symmetric_difference(set2)`
  - `set1 ^ set2`



Mengen - Methoden und Operatoren (4)

- alle Mengen-Operationen gibt es auch als ‘update‘-Methode / Operator
- kein Rückgabewert, in `set1` wird das resultierende Set behalten:
  - `set1.difference_update(set2)`
    `set1 -= set2`
  - `set1.symmetric_difference_update(set2)`
    `set1 ^= set2`
  - `set1.intersection_update(set2)`
    `set1 &= set2`
  - `set1 |= set2`

Unveränderliche Mengen: *frozenset*

- es gibt eine unveränderliche Mengenvariante, das `frozenset`
- funktioniert wie set: `fs = frozenset(sammelt)`
- aber: alle Methoden, die Elemente hinzufügen, Löschen oder verändern sind verboten (`add`, `remove`, `discard`, alle `update`-Methoden)
- Alle anderen Methoden und Operatoren funktionieren wie bei `set` (und geben ggf. `frozenset` statt `set` zurück)

Initialisierung von Listen, Mengen etc.

- die Sammeltypen, die keine Wörterbücher sind (:)) kann man direkt ineinander konvertieren
- das geht mit `typname(sammel_instanz)` - siehe Mengen

```
> menge = set([1,2])
> liste = list(menge)
> tupel = tuple(menge)
> tupel2 = tuple(liste)
> menge2 = set(liste *5)
...
```


Wörterbücher: `dict` (Dictionaries, Maps)

- Wörterbücher bilden (eindeutige) Schlüssel auf Werte ab; Schlüssel müssen einen unveränderlichen Typ haben
- Zugriff auf Werte über die Schlüssel
- Beispiel: ein Telefonbuch

```
> tel = {‘Mueller‘: 7234, ‘Meier‘: 8093}
> tel[‘Meier‘]
8093
> tel[‘Schmidt‘] = 2104
> tel
{‘Mueller‘: 7234, ‘Meier‘: 8093, ‘Schmidt‘: 2104}
```

Wörterbücher als Literale

- `{}` ist ein leeres Dictionary (!), genau wie `dict()`
- einige Alternative Schreibweisen mit dem gleichen Ergebnis:

```
> tel = {‘Mueller‘: 7234, ‘Meier‘: 8093}
```

```
> tel = dict([[‘Mueller‘, 7234], [‘Meier‘,8093]])
```

```
> tel = dict([(‘Mueller‘, 7234), (‘Meier‘, 8093)])
```

```
> tel = dict(Mueller=7234, Meier=8093)
```

nur mit gültigen Variablennamen als Schlüssel

Wörterbücher - Schlüssel (1)

- Schlüssel müssen unveränderliche Werte haben (siehe Mengen)
- `frozenset` ist demnach erlaubt (auch in Mengen)

```
> tel = {}
> tel[['Peter','Sophie']] = 7473
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type
```

```
> tel[frozenset(['Peter','Sophie'])] = 7473
> tel
{frozenset(['Peter','Sophie']):7473}
```

Wörterbücher - Schlüssel (2)

- Schlüsel, die beim Vergleich mit „==“ `True` ergeben, gelten als gleich
- wenn man einen Schlüssel belegt, der schon im Dictionary steht, kriegt er den neuen Wert (der alte wird gelöscht)

```
> tel[‘Peter‘] = 7473
> tel[‘Peter‘] = 9999
> tel
{‘Peter‘:9999}
```

- Vorsicht: `1` und `1.0` sind somit der gleiche Schlüssel

Wörterbücher - Methoden (1)

- Test, ob ein Schlüssel `key` in `dict` existiert:
  - `key in dict`
- Löschen eines Schlüssel-Wert-Paares (`key:value`):
  - `del dict[key]` (gibt nichts zurück)
  - `dict.pop(key)` (gibt `value` zurück)
- Setzen des Schlüssels `key` auf den Wert `value`, falls `key` noch nicht existiert:
  `dict.setdefault(key,value)`
  (wenn `key` existiert, wird der alte Wert von `key` zurückgegeben, sonst `value`)

Wörterbücher - Methoden (2)

- `dict1` mit Werten aus `dict2` ergänzen
  `dict1.update(dict2)`
  (Schlüssel, die in beiden sind, bekommen den Wert aus `dict2`)

- „View“ aller Schlüssel: `dict.keys()`
- „View“ aller Werte: `dict.values()`
- „View“ aller Schlüssel-Wert-Paare: `dict.items()`

Vorsicht: die Reihenfolge ist hier nicht deterministisch! Einzige Garantie: zwei Aufrufe hintereinander auf dem gleichen System ohne Veränderung von `dict` liefern die gleiche Reihenfolge, korrespondierend bei Schlüssel und Werten.

Wörterbücher -“Views“

- Views sehen z.B. so aus:

```
>>> map
{'a': 1, 'l': 3, 'o': 4}
>>> map.keys()
dict_keys(['a', 'l', 'o'])
```

- sie spiegeln jeweils den aktuellen Zustand des Dictionaries
- wir betrachten sie als Sammeltypen, die wir nicht verändern können (nicht als *unveränderlich*)
- zur weiteren Verarbeitung (wenn nötig) geht z.B.
  `liste = list(map.keys())`

Listen - Tupel - Mengen?

- feste Reihenfolge, alle Methoden zum verändern von Elementen: `list`
- feste Reihenfolge, keine Methoden oder Manipulation (unveränderlich): `tuple`
- keine feste Reihenfolge, manipulierbar: `set` (viel effizienter für Zugehörigkeits-Tests als Listen)
- unveränderliche Mengen: `frozenset`
- unmanipulierbare Typen als Schlüssel (in `dict`) und Mengenelemente (in `set` und `frozenset`)

`for`

- `S` ist ein Sammeltyp
- es wird über alle Elemente in `S` iteriert
- `i` ist das aktuell betrachtete Element
- bei jeder Iteration wird `block` ausgeführt
- `break`, `continue` und `else` funktionieren wie bei `while`

```
for i in S:
  block
```

```
> liste = [1,‘a‘,True]
> for i in liste:
..   print(i)
..
1
‘a‘
True
```

for

- Durchschnitt aller Listenelemente mit `for`:

```
def durchschnitt(liste):
  ergebnis = 0.0
  for zahl in liste:
    ergebnis += zahl
  return ergebnis / len(liste)
```

- Alphabetisch geordnete Schlüssel-Wert-Paare:

```
def sortedprint(map):
  schluessel = sorted(map.keys())
  for key in schluessel:
     print(str(key) + ':' + str(map[key]))
```

`sorted(map)` und `sorted(map.keys())` sind äquivalent

for mit Wörterbüchern

- auch über die „View"-Objekte der Wörterbücher (`keys`,`items`,`values`) kann man mit `for` iterieren
- oft will man über alle Paare in einem Wörterbuch iterieren, dank „sequence unpacking" geht das kurz so:

```
def oneLineOerentry(map):
  for key, val in map.items():
    print(str(key) + ':' + str(val))
```

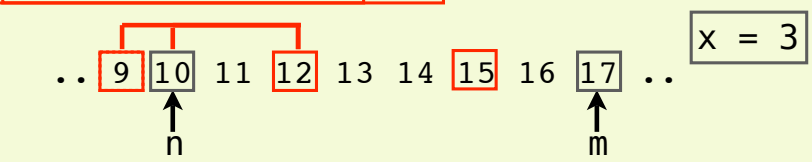
Exkurs: `range`

- Um Listen mit aufeinanderfolgenden Zahlen zu erzeugen, gibt es den Typ `range`
- `range` gibt nicht (mehr) direkt eine Liste zurück, sondern einen *Iterator*-ähnlichen Sammeltyp
- Iteratoren kann man mit For-Schleifen benutzen
  - `range(m)` entspricht den Elementen `[0,1,...,m-1]`
  - `range(n,m)` ≈ `[n, n+1, ..., m-1]`
  - `range(n,m,k)` macht dabei `k`-er Schritte (vgl. Slicing)

Vielfache berechnen mit `for`

nächstes Vielfaches von x nach n-1

n + 1 - (n - 1) % x




```
def vielfache(x,n,m):
 start = (n-1) + x -((n-1)%x)
 for i in range(start,m,x):
   print(i)
```

```
> vielfache(3,11,24)
12
15
18
21
```

Zusammenfassung

- Funktionen
- Rekursion
- Sammeltypen: List, Tupel, Mengen, Wörterbücher
- neue Kontrollstruktur: for-Schleife