Programmierkurs Python I

Michaela Regneri & Stefan Thater
Universität des Saarlandes
FR 4.7 Allgemeine Linguistik (Computerlinguistik)

Winter 2010/11

Übersicht

- Kurze Wiederholung: `while`
- Sammeltypen (kurz & knapp)
  - mehr zu Sammeltypen in der nächsten Sitzung
- Schleifen: for
- Funktionen

while

1. Der Ausdruck `expr` wird ausgewertet
2. Trifft er zu, wird `block` ausgeführt
   - Danach gehe zu 1
3. Sonst wird der Programmfluss hinter der Schleife fortgesetzt.

```
while expr:
  block
...
```


break & continue

- Die `break`-Anweisung
  - verlässt die aktuelle Schleife
  - (ohne die Bedingung auszuwerten)
- Die `continue`-Anweisung
  - überspringt den Rest des Blocks
  - wertet die Bedingung neu aus
  - und setzt ggf. die Schleife fort.

while – break – else

- Schleifen können einen `else` Block haben.
- Der `else` Block wird ausgewertet sobald die Bedingung der Schleife zu falsch auswertet, ...
- aber nicht, wenn die Schleife mit `break` verlassen wurde.

```
while bedingung:
   ...
   if irgendwas:
     break
   ...
else:
   ...
```


Beispiel: Primzahlen von 2 ... 100

```
n = 2
while n < 100:
   m = 2
   while m < n:
      if n % m == 0:
         break
      m += 1
   else:
      print(n, 'ist eine Primzahl')
   n += 1
```

Sammeltypen

- Sammeltypen dienen als „Container“ für beliebige Werte
- Sammeltypen in Python:
  - Listen – Sammlung von Elementen, feste Reihenfolge
  - Strings – Listen von Zeichen (unveränderlich)
  - Tupel – wie Listen, aber unveränderlich
  - Mengen – ungeordnete Sammlung von Elementen
  - Wörterbücher – Abbildungen von Schlüssel auf Werte
  - ...
- Alle Sammeltypen unterstützen:
  - len(x) gibt Anzahl der Elemente in x

Listen (list)

- Eine Liste ist eine geordnete Sammlung von Werten
  - `liste = ['a', 'Hallo', 1, 3.0, [1, 2, 3]]`
- Die Listenelemente können verschiedene Typen haben
- Zugriff auf Listenelemente:
  - `liste[0]` ⇒ `'a'`
  - `liste[-1]` ⇒ `[1,2,3]`
  - `liste[-1][1]` ⇒ `2`
  - `liste[5]` ⇒ `IndexError: list index out of range`

Tupel (tuple)

- Ähnlich wie Listen
  - `tup = ('a', 1, 1.2)`
  - `tup = 'a', 1, 1.2`
  - `tup = tuple(['a', 1, 1.2])`
- Spezielle Syntax für Tupel mit einem Element:
  - `tup = ('a',)`
- Hauptunterschied zu Listen:
  - Das Hinzufügen, Löschen oder Austauschen von Elementen eines Tupels ist nicht möglich
- Zugriff auf Elemente wie bei Listen

for-Schleifen

- In einer for-Schleife wird über alle Elemente in seq iteriert
  - `seq` ein beliebiger Sammeltyp-Wert
  - zum Beispiel: Liste, Tupel, String, ...
- In jedem Iterationsschritt wird $\texttt{block}_1$ ausgeführt
- `i` ist das aktuell betrachtete Element
  - Reihenfolge: seq[0], seq[1], …
  - (keine feste Reihenfolge bei Mengen, Wörterbüchern)
- `break`, `continue` und `else` funktionieren wie bei `while`

```
for i in seq:
  block1
[else:
  block2]
```

for-Schleifen (Beispiele)

```
>>> for c in "Python":
...   print(c)

P
y
t
h
o
n
```

```
>>> a = [1, 2, 3, 4]
>>> b = 0
>>> for x in a:
...   b += x
>>> b
10
>>> x
4
```


for-Schleifen mit „sequence unpacking“

- Oft sind die Elemente von Listen wieder Sammeltyp-Werte
- Dank „sequence unpacking“ kann man auf die Elemente der einzelnen Listenelemente bequem zugreifen:

```
>>> tel = [('Müller', 7234), ('Meier', 8093)]
>>> for k, v in tel:
...   print(k + ': ' + str(v))

Müller: 7234
Meier: 8093
```

- Beachte: beim sequence unpacking müssen beide Sammeltyp-Werte die gleiche Länge haben

range

- Die eingebaute Funktion `range` erzeugt Folgen von aufeinanderfolgenden Zahlen (`int`)
- `for x in range(10): print(x)`
  - Ausgabe: 0, 1, 2, ..., 9
- `for x in range(3, 10): print(x)`
  - Ausgabe: 3, 4, 5, ..., 9
- `for x in range(3, 10, 2): print(x)`
  - Ausgabe: 3, 5, 7, 9

Funktionen – Übersicht

- Funktionen elementar
- Namensräume
- Rekursion
- Syntaktischer Zucker:
  - Variable Argumentlisten
  - Default-Werte
  - Schlüsselworte

Funktionen

- Funktionen sind wiederverwendbare Code-Blöcke die „Unterprogramme“ implementieren
- Funktionen strukturieren das Gesamtproblem in mehrere kleinere Teilprobleme
- Die Funktionsdefinition entspricht der Zuweisung einer Funktion an eine Variable
- Wird eine Funktion aufgerufen, wird der Code der Funktion ausgeführt

Funktionen

```
>>> def hello(someone):
...     """Say hello to someone"""
...     print("Hello, " + someone + "!")
...
>>> hello
<function hello at 0x7fb3907a2a68>
>>> hello("Michaela")
Hello, Michaela!
>>> help(hello)
[...] Say hello to someone
```

Syntax

def <name>(var_1, ..., var_n):
    [<documentation string>]
    block

Parameter und lokale Variablen

- Funktionen können Argumente (Parameter) haben.
  - Parameter sind lokale Variablen, deren Wert beim Aufruf der Funktion bestimmt wird.
- Zuweisungen innerhalb einer Funktionsdefinition „erzeugen“ lokale Variablen
  - Wenn einer Variablen ein Wert zugewiesen wird, wird die Variable als lokale Variable aufgefasst.
- Lokale Variablen = nur innerhalb der Funktion „sichtbar“

Funktionen

```
>>> def fak(n):
...     result = 1
...     for i in range(1, n + 1):
...         result *= i
...     return result
...
>>> fak(4)
24
```

lokale Variablen

- Die `return` Anweisung verlässt die Funktion …
- und liefert den Wert des Arguments als Resultat des Funktionsaufrufs.

Namensräume

- Ein Namensraum (Namespace) ist eine Abbildung von Bezeichnern (Namen) auf Objekte
- Namen in verschiedenen Namensräumen können auf verschiedene Objekte referieren
  - Eine lokale Variable „x“ und eine globale Variable „x“ sind verschiedene Variablen
- Namensräume können geschachtelt sein.

Namensräume

- Bei Funktionsaufrufen wird ein neuer lokaler Namensraum erzeugt
  - Lokale Variablen existieren (nur) im lokalen Namensraum
  - (Die Funktions-Parameter sind auch lokale Variablen)
- Beim Verlassen der Funktion wird der Namensraum wieder gelöscht bzw. „vergessen“
- Bei Rekursion hat jeder rekursive Aufruf der Funktion seinen eigenen Namensraum

Beispiel

```
>>> n = 17
>>> def add_to_n(x):
...     return x + n
...
>>> add_to_n(10)
27
>>> x
NameError: name 'x' is not defined
```


Beispiel

```
>>> n = 17
>>> def add_to_n(x):
...     n = 27
...     return x + n
...
>>> add_to_n(10)
37
>>> n
17
```

Funktionen

- Innerhalb von Funktionen haben wir Zugriff auf
  - die Parameter
  - lokale Variablen
  - globale Variablen - aber nur „lesend“
- Außerhalb der Funktion sind die lokalen Variablen nicht sichtbar

Beispiel

```
>>> counter = 17
>>> def add1():
...     counter = counter + 1
...
>>> add1()
UnboundLocalError: local variable 'counter' referenced
before assignment
```

Beispiel

```
>>> counter = 17
>>> def add1():
...     global counter
...     counter = counter + 1
...
>>> counter
17
>>> add1()
>>> counter
18
```

Die „global“ Anweisung deklariert Variablen als global.

Rekursion

- Funktionen können andere Funktionen aufrufen
- Funktionen können auch sich selbst aufrufen
  - das nennt man „Rekursion“
- Rekursion ist ein mächtiges Werkzeug, mit dem man viele Algorithmen elegant ausdrücken kann
- Aber Vorsicht: man muss darauf achten, dass die Rekursion irgendwo endet!

Beispiel: Fakultätsfunktion

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

Rekursionsabbruch für 0 und 1

Rekursion mit absteigendem n

Beispiel: Fibonacci-Zahlen

- Die Fibonacci-Folge ist eine unendliche Folge von Zahlen, bei der sich die jeweils folgende Zahl durch Addition der beiden vorherigen Zahlen ergibt:
  - 0, 1, 1, 2, 3, 5, 8, 13, ...

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

Funktionen – Übersicht

- Funktionen elementar
- Namensräume
- Rekursion
- Syntaktischer Zucker:
  - Variable Argumentlisten
  - Default-Werte
  - Schlüsselworte

Variable Argument-Listen

```
>>> def summe(*args):
...     result = 0
...     for item in args:
...         result += item
...     return result
...
>>> summe(1)
1
>>> summe(1,2)
3
>>> summe(1,2,3)
6
```

Unpacking

```
>>> def add(x, y):
...     return x + y
...
>>> pair = (1, 2)
>>> add(pair[0], pair[1])
3
>>> add(*pair)
3
```


Optionale Argumente

```
>>> def summe(seq, start = 0):
...     for i in seq:
...         start += i
...     return start
...
>>> summe([1,2,3,4])
10
>>> summe([1,2,3,4], 22)
32
>>> summe([1,2,3,4], start=22)
32
```

Achtung!

```
>>> def summe(seq, start = [0]):
...     for i in seq:
...         start[0] += i
...     return start[0]
...
>>> summe([1,2,3,4])
10
>>> summe([1,2,3,4])
20
>>> summe([1,2,3,4])
30
```

- Default-Werte werden evaluiert, wenn die Funktion definiert wird,
- und nicht, wenn die Funktion aufgerufen wird!

Optionale Argumente, etc.

- Mehr zu optionalen Argumenten, variablen Argumentlisten, ... siehe
  - http://docs.python.org/py3k/tutorial/index.html
  - Abschnitt 4.7

Zusammenfassung

- Kontrollstruktur: for-Schleife
- Funktionen
- Rekursion