

Informatik I

4. Funktionen: Aufrufe und Definitionen

Bernhard Nebel

Albert-Ludwigs-Universität Freiburg

25. Oktober 2013

Informatik I

25. Oktober 2013 — 4. Funktionen: Aufrufe und Definitionen

- 4.1 Funktionsaufrufe
- 4.2 Mathematische Funktionen
- 4.3 Funktionsdefinitionen
- 4.4 Variablengültigkeitsbereich
- 4.5 Rückgabewerte

Fkt.-Aufrufe

4.1 Funktionsaufrufe

Fkt.-Aufrufe

Funktionsaufrufe

- ▶ Innerhalb von Programmiersprachen ist eine **Funktion** ein Programmstück (meistens mit einem Namen versehen).
- ▶ Normalerweise erwartet eine Funktion **Argumente** und gibt einen **Funktionswert** (oder *Rückgabewert*) zurück.
- ▶ type-Funktion:

Python-Interpreter

```
>>> type(42)
<class 'int'>
```

- ▶ Funktion mit variabler Anzahl von Argumenten und ohne Rückgabewert (aber mit Seiteneffekt): **print**
- ▶ Funktion ohne Argumente und ohne Rückgabewert: **exit**

Standardfunktionen: Typen-Konversion

Mit den Funktionen `int`, `float`, `complex` `str` kann man „passende“ Werte in den jeweiligen Typ umwandeln. Umwandlung nach `int` durch „Abschneiden“.

Python-Interpreter

```
>>> int(-2.6)
-2
>>> int('vier')
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() ...
>>> complex('42')
(42+0j)
>>> float(4)
4.0
>>> str(42)
'42'
```

Standardfunktionen: Zeichen-Konversion

Mit den Funktionen `chr` und `ord` kann man Zahlen in **Unicode-Zeichen** und umgekehrt umwandeln, wobei in Python Zeichen identisch mit einbuchstabigen Strings sind:

Python-Interpreter

```
>>> chr(42)
'*'
>>> chr(255)
'ÿ'
>>> ord('*')
42 >>> ord('**')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ord() expected a character, but string of length
2 found
```

4.2 Mathematische Funktionen

Mathematische Funktionen: Das Math-Modul

- ▶ Natürlich wollen wir Funktionen wie `sin` verwenden. Die muss man in Python aber erst durch **Importieren** des **Mathematik-Moduls** bekannt machen.
- ▶ Danach können wir die Teile des Moduls durch Voranstellen von `math.` nutzen (**Punktschreibweise**):

Python-Interpreter

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sin(1/4*math.pi)
0.7071067811865475
>>> math.sin(math.pi)
1.2246467991473532e-16
>>> math.exp(math.log(2))
2.0
```

Mathematische Funktionen: Direkt importieren

- ▶ Die Punktschreibweise verhindert **Namenskollisionen**, ist aber umständlich
- ▶ Mit `from module import name` kann ein Name direkt importiert werden.
- ▶ `from module import *` werden alle Namen direkt importiert.

Python-Interpreter

```
>>> from math import pi
>>> pi
3.141592653589793
>>> from math import *
>>> cos(pi)
-1.0
```

4.3 Funktionsdefinitionen

Neue Funktionen definieren

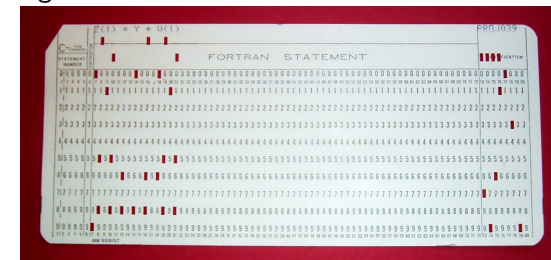
- ▶ Mit dem Schlüsselwort `def` kann man eine neue Funktion einführen.
- ▶ Nach `def` kommt der **Funktionsname** gefolgt von der Argumentliste und dann ein Doppelpunkt.
- ▶ Nach dem **Funktionskopf** gibt der Python-Interpreter das **Funktionsprompt**-Zeichen `...` aus.
- ▶ Dann folgt der **Funktionsrumpf**: *Gleich weit eingerückte Anweisungen*, z.B. Zuweisungen oder Funktionsaufrufe:

Python-Interpreter

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay")
...     print("I sleep all night and I work all day")
...
>>>
```

Einrückungen in Python

- ▶ Im Gegensatz zu fast allen anderen Programmiersprachen (außer z.B. FORTRAN, Miranda, Haskell), sind **Einrückungen** am Zeilenanfang bedeutungstragend.



- ▶ In Python ist gleiche Einrückung = zusammen gehöriger Block von Anweisungen
 - ▶ In den meisten anderen Programmiersprachen durch Klammerung `{ }` oder klammernde Schlüsselwörter.
 - ▶ Wie viele Leerzeichen sollte man machen?
- **PEP8**: 4 Leerzeichen pro Ebene (keine Tabs nutzen)

Selbst definierte Funktionen nutzen

- ▶ Funktionsnamen müssen den gleichen Regeln folgen wie Variablennamen.
- ▶ Tatsächlich verhalten sich Funktionsnamen wie Variablennamen und haben einen entsprechenden Typ.
- ▶ Man kann eigene Funktionen wie Standardfunktionen aufrufen

Python-Interpreter

```
>>> print(print_lyrics)
<function print_lyrics at 0x100520560>
>>> type(print_lyrics)
<class 'function'>
>>> print_lyrics()
I'm a lumberjack, and I'm okay
I sleep all night and I work all day
>>> print_lyrics = 42
```

Definierte Funktionen in Funktionsdefinitionen

Was passiert hier?

Python-Interpreter

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay")
...     print("I sleep all night and I work all day")
...
>>>
>>> def repeat_lyrics():
...     print_lyrics()
...     print_lyrics()
...
>>> repeat_lyrics()
I'm a lumberjack ...
Was wird hier exakt ausgeführt?
```

Argumente und Parameter

- ▶ Auch definierte Funktionen brauchen manchmal *Argumente*.
- ▶ Bei der Definition gibt man **Parameter** an, die beim Aufruf durch die Argumente ersetzt werden.

Python-Interpreter

```
>>> michael = 'baldwin'
>>> def print_twice(bruce):
...     print(bruce)
...     print(bruce)
...
>>> print_twice(michael)
baldwin
baldwin
>>> print_twice('Spam ' * 3)
Spam Spam Spam
Spam Spam Spam
```

Funktionen als Argumente

- ▶ Wir können Funktionen wie andere Werte als Argumente übergeben.

Python-Interpreter

```
>>> def do_twice(f):
...     f()
...     f()
...
>>> do_twice(print_lyrics)
I'm a lumberjack, and I'm okay
I sleep all night and I work all day
I'm a lumberjack, and I'm okay
I sleep all night and I work all day
```

4.4 Variablen gültigkeitsbereich

Gültigkeitsbereich von Variablen und Parametern

- ▶ Parameter sind nur innerhalb der Funktion **sichtbar/gültig**.
- ▶ Lokal eingeführte Variablen ebenfalls.
- ▶ Ihre Lebenszeit ist auf den Aufruf beschränkt.

Python-Interpreter

```
>>> def cat_twice(part1, part2):
...     cat = part1 + part2
...     print_twice(cat)
...
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
>>> cat
NameError: name 'cat' is not defined
```

Stapel diagramme

- ▶ Entsprechend zu den Zustandsdiagrammen kann man die Variablenbelegungen in **Stapel diagrammen** visualisieren:

<module>

```
line1 → 'Bing tiddle '
line2 → 'tiddle bang'
```

cat_twice

```
part1 → 'Bing tiddle '
part2 → 'tiddle bang'
cat   → 'Bing tiddle tiddle bang'
```

print_twice

```
bruce → 'Bing tiddle tiddle bang'
```

Traceback

- ▶ Tritt bei der Ausführung einer Funktion ein Fehler auf (z.B. Zugriff auf die nicht vorhandene Variable `cat` in `print_twice`, dann gibt es ein **Traceback** (entsprechend zu unserem Stapeldiagramm):

Python-Interpreter

```
>>> cat_twice(line1, line2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in cat_twice
  File "<stdin>", line 3, in print_twice
NameError: global name 'cat' is not defined
```

Globale Variablen

- ▶ Man sollte möglichst nur lokale Variable und Parameter in einer Funktion verwenden.
- ▶ Manchmal möchte man aber auch **globale Variablen** einsetzen (z.B. zur globalen Moduseinstellung oder für Zähler).
- ▶ Dafür gibt es das Schlüsselwort `global`.

Python-Interpreter

```
>>> counter = 0
>>> def inc():
...     global counter
...     counter = counter + 1
...
>>> inc()
>>> counter
1
```

4.5 Rückgabewerte

Funktionen mit und ohne Rückgabewert

- ▶ Funktionen können einen Wert zurückgeben, wie z.B. `chr` oder `sin`.
- ▶ Einige Funktionen haben keinen Rückgabewert, weil sie nur einen (Seiten-)Effekt verursachen sollen, wie z.B. `inc` und `print`.
- ▶ Tatsächlich geben diese den speziellen Wert `None` zurück.

Python-Interpreter

```
>>> result = print('Bruce')
Bruce
>>> result
>>> print(result)
None [≠ der String 'None']
```

Einen Wert zurück geben

- ▶ Wollen wir einen Wert zurück geben, müssen wir das Schlüsselwort **return** benutzen.

Python-Interpreter

```
>>> def sum3(a, b, c):
...     return a + b + c
...
>>> sum3(1, 2, 3)
6
```