

Programmieren in Python

3. Variablen, Funktionen und Bedingungen

Malte Helmert

Albert-Ludwigs-Universität Freiburg

KI-Praktikum, Sommersemester 2009

1 / 35

Variablen, Funktionen und Bedingungen

Bisher sind wir über die Funktionen eines Taschenrechners nicht hinaus gekommen. Damit sich daran etwas ändert, brauchen wir auf jeden Fall Variablen und Funktionen. Im Einzelnen behandeln wir:

- ▶ Variablen
- ▶ Funktionen: `def`, `return`
- ▶ Bedingungen: `if`, `while`, `bool`

2 / 35

Variablen, Funktionen und Bedingungen

Bisher sind wir über die Funktionen eines Taschenrechners nicht hinaus gekommen. Damit sich daran etwas ändert, brauchen wir auf jeden Fall Variablen und Funktionen. Im Einzelnen behandeln wir:

- ▶ **Variablen**
- ▶ Funktionen: `def`, `return`
- ▶ Bedingungen: `if`, `while`, `bool`

3 / 35

Variablen

- ▶ Variablen und Zuweisungen sehen in Python nicht sonderlich überraschend aus:

```
Python-Interpreter
>>> spam = 111 * 111
>>> spam
12321
>>> egg = spam * spam
>>> egg
151807041
>>> spam = egg * egg
>>> spam
23045377697175681L
```

4 / 35

Variablen: Details

- ▶ Für Variablenamen gelten dieselben Regeln wie in C: Erlaubt sind Buchstaben (ASCII, keine Umlaute etc.) und Unterstriche sowie Ziffern. Das erste Zeichen darf keine Ziffer sein.
- ▶ Variablen müssen nicht deklariert werden, sondern werden ins Leben gerufen, sobald ihnen erstmals ein Wert zugewiesen wird. Im Gegensatz zu Perl oder PHP können Variablen nicht verwendet werden, bevor ihnen ein Wert zugewiesen wurde:

Python-Interpreter

```
>>> spam = 3
>>> print spam
3
>>> print 10 + egg
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'egg' is not defined
```

5 / 35

Variablen: Typprüfung (1)

- ▶ Python ist zwar eine stark getypte Sprache, aber im Gegensatz zu Sprachen wie C und Java sind nicht *Variablen*, sondern *Ausdrücke* getypt. Das bedeutet, dass derselben Variable nacheinander Werte mit unterschiedlichen Typen zugewiesen werden können:

Python-Interpreter

```
>>> spam = 1+7j
>>> print spam
(1+7j)
>>> spam = 100L
>>> print spam
100
>>> spam = "spam, spam, spam, spam"
>>> print spam
spam, spam, spam, spam
```

6 / 35

Variablen: Typprüfung (2)

- ▶ Ausdrücke sind aber getypt, und Typfehlern werden erkannt:

Python-Interpreter

```
>>> print "eins" + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
>>> print 1j < 4
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot compare complex numbers using <, <=, >, >=
>>> print "Gambolputty"[2.0]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: string indices must be integers
```

7 / 35

Variablen: Tuple Unpacking (1)

- ▶ Man kann mehrere Werte gleichzeitig zuweisen:

Python-Interpreter

```
>>> spam, egg = 7, 13
>>> print spam
7
>>> print egg
13
```

- ▶ Diesen Vorgang bezeichnet man als „tuple unpacking“ – mehr zu Tupeln später.

8 / 35

Variablen: Tuple Unpacking (2)

- ▶ Bei simultanen Zuweisungen werden alle zugewiesenen Werte berechnet, bevor der erste Wert zugewiesen wird, was folgenden Trick ermöglicht:

Python-Interpreter

```
>>> spam = "spam"
>>> egg = "egg"
>>> spam, egg = egg, spam
>>> print spam
egg
>>> print egg
spam
```

9 / 35

Variablen: Die Variable _

- ▶ Im Interpreter (und nur dort!) hat die Variable `_` eine besondere Bedeutung. Sie enthält den Wert des letzten "nackten Ausdrucks", der berechnet wurde:

Python-Interpreter

```
>>> 2 ** 10
1024
>>> _
1024
>>> _ * _
1048576
>>> _ * _
1099511627776L
```

10 / 35

Variablen, Funktionen und Bedingungen

Bisher sind wir über die Funktionen eines Taschenrechners nicht hinaus gekommen. Damit sich daran etwas ändert, brauchen wir auf jeden Fall Variablen und Funktionen. Im Einzelnen behandeln wir:

- ▶ Variablen
- ▶ Funktionen: `def`, `return`
- ▶ Bedingungen: `if`, `while`, `bool`

11 / 35

Funktionen

Um vernünftig programmieren zu können, benötigen wir auf jeden Fall noch Funktionen. Hier sehen wir eine der Besonderheiten von Python:

Python-Interpreter

```
>>> def triple(egg):
...     return 3 * egg
...
>>> print triple(10)
30
>>> def double(spam):
...     return 2 * spam
File "<stdin>", line 2
    return 2 * spam
    ^
IndentationError: expected an indented block
```

12 / 35

Einrücken!

- ▶ Python markiert die Blockstruktur eines Programs durch **Einrücken**, nicht durch Klammerpaare oder Schlüsselwörter.
- ▶ Die Regeln sind ganz einfach: Alles, was zu der Funktion gehört, muss gegenüber der Funktionsdefinition eingerückt sein.
- ▶ Anders ausgedrückt: Man rückt einfach so ein, wie man es in einer anderen Programmiersprache nach den üblichen Konventionen.

Dieses Feature ist so charakteristisch für Python, dass es dazu ein passendes T-Shirt gibt:

Python — Programming the way Guido indented it.

13 / 35

Einrücken im interaktiven Interpreter

- ▶ Wie in den Beispielen gesehen, schaltet der interaktive Interpreter den normalen Prompt `>>>` auf den *Fortsetzungsprompt* `. . .` um, wenn man sich in einem eingerückten Block befindet.
- ▶ Der eingegebene Code wird erst verarbeitet, wenn der eingerückte Block beendet ist (anders ginge es ja auch nicht).
- ▶ Um dem Interpreter mitzuteilen, dass keine weiteren eingerückten Zeilen mehr folgen und er den eingegebenen Code jetzt verarbeiten soll, gibt man eine Leerzeile ein.
- ▶ Diese Leerzeilen sind *nur* im interaktiven Interpreter erforderlich!

14 / 35

Achtung: Tabs und Spaces

- ▶ Python versteht zum Einrücken sowohl Leerzeichen als auch Tabulatorzeichen.
- ▶ Solange man die beiden nicht vermischt, gibt es keine Probleme.
- ▶ Tut man dies aber doch (böse! böse!), muss sich der benutzte Editor an den Standard für Tabulatoren halten (Tabstops an den Positionen 1, 9, 17, ...).
- ▶ Am besten sorgt man dafür, dass der Editor gar keine Tabulatorzeichen in die Datei schreibt.
- ▶ Ruft man `python` mit der Option `-t` auf, wird vor „gefährlichen“ Dateien gewarnt.

15 / 35

Wo wir schon mal bei der Formatierung sind . . .

- ▶ Wie wohl schon aufgefallen ist, wird auch das *Ende der Anweisung* in Python durch Formatierung bestimmt.
- ▶ Genauer: Das Zeilenende beendet die Anweisung.
- ▶ Manchmal möchte man das nicht:

```
ein_langer_name = noch_ein_langer_name +  
                  ein_noch_laengerer_name
```

Dies führt nach Verarbeitung der ersten Zeile zu einer Fehlermeldung.

16 / 35

Lange Zeilen (2)

Es gibt zwei Methoden zur Abhilfe:

- ▶ Endet eine Zeile mit einem Backslash, wird sie — wie in C — mit der folgenden Zeile vereinigt:

```
ein_langer_name = noch_ein_langer_name + \  
                  ein_noch_laengerer_name
```

- ▶ Wenn eine schließende Klammer (rund, eckig oder geschweift) aussteht, beenden Zeilenenden die Anweisung *nicht*:

```
ein_langer_name = (noch_ein_langer_name +  
                  ein_noch_laengerer_name)
```

Es gilt als guter Stil, die erste Variante zu vermeiden und im Zweifelsfall zusätzliche Klammern einzuführen, um die zweite Variante verwenden zu können.

Zurück zu den Funktionen

Python-Interpreter

```
>>> def triple(egg):  
...     return 3 * egg  
...  
>>> print triple(10)  
30  
>>> def double(spam):  
...     return 2 * spam  
...  
>>> print double(5.5)  
11.0
```

Funktionen: Details

- ▶ Funktionen werden mit dem Schlüsselwort `def` eingeleitet.
- ▶ Funktionen können eine variable Zahl an Argumenten akzeptieren und Default-Argumente haben — mehr dazu später.
- ▶ Werte werden mit dem Schlüsselwort `return` zurückgegeben.
- ▶ Funktionen haben immer ein Resultat.
 - ▶ Wird nicht explizit ein Wert zurückgegeben, ist das Resultat der spezielle Wert `None`, der eine (die einzige) Instanz der Klasse `NoneType` ist.
 - ▶ `None` spielt in Python die Rolle von `NULL` in C/C++, `null` in Java, `nil` in LISP und `void` in C, C++ und Java.

Funktionen: Beispiele

Python-Interpreter

```
>>> def product(x, y, z):  
...     return x * y * z  
...  
>>> print product(5, 6, 7)  
210  
>>> def sing():  
...     print "spam, spam, spam, spam"  
...  
>>> sing()  
spam, spam, spam, spam  
>>> print sing()  
spam, spam, spam, spam  
None
```

Funktion: Polymorphie (1)

- ▶ Python-Funktionen sind von Haus aus polymorph:

Python-Interpreter

```
>>> def double(anything):
...     return 2 * anything
...
>>> print double(10.2)
20.4
>>> print double("badger")
badgerbadger
```

21 / 35

Funktion: Polymorphie (2)

- ▶ Typfehler werden zur Laufzeit erkannt:

Python-Interpreter

```
>>> def double(anything):
...     return 2 * anything
...
>>> print double(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 1, in double
TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
```

- ▶ Mehr zu Pythons Polymorphie unter dem Thema *duck typing*.

22 / 35

Funktionen mit mehreren Ergebnissen

- ▶ Funktionen können mehrere Resultate haben.
- ▶ Ein Beispiel dafür ist die eingebaute Funktion `divmod`:

Python-Interpreter

```
>>> print divmod(20, 6)
(3, 2)
```

- ▶ Auch hier greift der früher erklärte Tuple-Unpacking-Mechanismus:

Python-Interpreter

```
>>> x, y = divmod(20, 6)
>>> print x
3
>>> print y
2
```

- ▶ Mehr dazu später unter dem Thema *Tupel*.

23 / 35

Eingebaute Funktionen („builtins“)

- ▶ Ähnlich wie in Java sind in Python die meisten Funktionen in *Paketen* und *Modulen* untergebracht.
- ▶ Einige häufig benutzte Funktionen sind aber automatisch verfügbar — die sogenannten „builtins“.

Einige wichtige builtins:

- ▶ `abs(x)`
 - ▶ Absolutbetrag der Zahl x .
 - ▶ Funktioniert auch für komplexe Zahlen.
- ▶ `divmod(x, y)`
 - ▶ $x // y$ und $x \% y$ (liefert zwei Ergebnisse).
- ▶ `min(x, y, ...)`, `max(x, y, ...)`
 - ▶ Minimum bzw. Maximum der übergebenen Zahlen.
 - ▶ Variable Argumentzahl (mindestens 1).
- ▶ `pow(x, y[, z])`
 - ▶ Bei zwei Argumenten: Berechnet $x ** y$.
 - ▶ Bei drei Argumenten: Berechnet $(x ** y) \% z$.

24 / 35

Weitere eingebaute Funktionen: int, long & Co.

- ▶ Zu den Datentypen int, long, float und complex gibt es gleichnamige Funktionen, die ihr Argument in den entsprechenden Typ umwandeln.
- ▶ Die Funktionen int und long schneiden den Nachkommateil eines Fließkommarguments ab:

Python-Interpreter

```
>>> int(7.1)
7
>>> long(-7.1)
-7L
```

- ▶ int liefert ein long, falls das Argument für ein int zu groß ist.
- ▶ complex akzeptiert ein optionales zweites Argument, das den Imaginärteil angibt.
- ▶ Eigentlich handelt es sich nicht um „normale“ Funktionen, sondern eher um Konstruktoren der gleichnamigen Klassen — später mehr.

25 / 35

Eingebaute Funktionen: Beispiele

Python-Interpreter

```
>>> min(10, 7.0, -100L)
-100L
>>> divmod(100, 3)
(33, 1)
>>> pow(3, 1000000, 10)
1
>>> max()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: max expected 1 arguments, got 0
>>> float(10)
10.0
>>> complex(_)
(10+0j)
```

26 / 35

Variablen, Funktionen und Bedingungen

Bisher sind wir über die Funktionen eines Taschenrechners nicht hinaus gekommen. Damit sich daran etwas ändert, brauchen wir auf jeden Fall Variablen und Funktionen. Im Einzelnen behandeln wir:

- ▶ Variablen
- ▶ Funktionen: def, return
- ▶ Bedingungen: if, while, bool

27 / 35

Abfragen mit if und else

Python-Interpreter

```
>>> def factorial(n):
...     if n <= 1:
...         return 1
...     else:
...         return n * factorial(n - 1)
...
>>> print factorial(10)
3628800
```

- ▶ Hoffentlich selbsterklärend...

28 / 35

Mehr zu if und else

- ▶ Neben if und else gibt es noch *else-if*-Blöcke, die man in Python als elif notiert. Die allgemeine Form ist dabei:

```
if <bedingung>:  
    <Code>  
elif <bedingung>:  
    <Code>  
elif <bedingung>:  
    <Code>  
...  
else:  
    <Code>
```

- ▶ elif- und else-Teile sind optional.

29 / 35

while-Schleifen

Python-Interpreter

```
>>> def factorial(n):  
...     result = 1  
...     i = 2  
...     while i <= n:  
...         result = result * i  
...         i = i + 1  
...     return result  
...  
>>> print factorial(10)  
3628800
```

- ▶ Hoffentlich selbsterklärend...
- ▶ while-Schleifen sind in Python vergleichsweise selten, da for (später) meistens eine bessere Alternative ist.

30 / 35

Mehr zu Bedingungen

- ▶ Für Bedingungen stehen die Vergleichsoperatoren aus C/C++ und Java bereit:
 - ▶ ==, !=: Gleichheit und Ungleichheit
 - ▶ <, <=, >, >=: Anordnung
- ▶ Verknüpfungen von Aussagen werden in Python als and, or und not geschrieben (nicht &&, ||, !).
- ▶ Vergleiche können wie in der Mathematik verkettet werden:
 $0 \leq x \leq 10$ ist also äquivalent zu $(0 \leq x) \text{ and } (x \leq 10)$.

31 / 35

Beispiel zu and, or und not

Python-Interpreter

```
>>> def is_probably_prime(n):  
...     if n == 2:  
...         return True  
...     elif n <= 1 or n % 2 == 0:  
...         return False  
...     elif n <= 20 and not (n == 9 or n == 15):  
...         return True  
...     else:  
...         return n % 3 != 0 and n % 5 != 0
```

32 / 35

Bedingungen: Der Typ bool

- ▶ Vergleiche können auch außerhalb von `if` und `while` auftreten:

Python-Interpreter

```
>>> print 1 + 1 == 2
True
>>> condition = 24 < 6 * 6 < 26
>>> condition
False
```

- ▶ Das Ergebnis eines Vergleichs ist entweder die Konstante `False` oder die Konstante `True`.
- ▶ Der Typ dieser Konstanten ist `bool`.

33 / 35

Mehr zu bool

- ▶ Wie in C++ entsprechen `True` und `False` den Zahlen 1 und 0, und man kann mit ihnen rechnen:

Python-Interpreter

```
>>> int(True)
1
>>> print True + True
2
```

34 / 35

Mehr zu bool (2)

- ▶ Ebenso wie `int`, `float` & Co. kann man `bool` als Funktion (bzw. Konstruktor) verwenden.
 - ▶ `bool(None)` ist `False`.
 - ▶ `bool(0)` ist `False` (ebenso für `0L`, `0.0` oder `0j`).
 - ▶ Ansonsten gilt `bool(x) == True`, falls `x` eine Zahl ist.
- ▶ Die Bedingung bei `if` und `while` muss nicht unbedingt ein `bool` sein. Ist sie es nicht, wird sie nach `bool` konvertiert.
Also könnten wir auch schreiben:

Python-Interpreter

```
>>> def factorial(n):
...     result = 1
...     while n:
...         result = result * n
...         n = n - 1
...     return result
```

35 / 35