Programmieren in Python
5. Mehr zu Strings & ein paar Worte zu Objekten

Robert Mattmüller

Albert-Ludwigs-Universität Freiburg

Handlungsplanungs-Praktikum
Wintersemester 2010/2011

Mehr zu Strings & ein paar Worte zu Objekten

Wir befassen uns mit den tausend Möglichkeiten, einen String zu notieren, erzeugen vernünftige Ausgaben und streifen den Begriff des Objekts.

Im Einzelnen:

- Kodierung
- String-Literale
- String-Interpolation
- Objekte und Methoden

Mehr zu Strings & ein paar Worte zu Objekten

Wir befassen uns mit den tausend Möglichkeiten, einen String zu notieren, erzeugen vernünftige Ausgaben und streifen den Begriff des Objekts.

Im Einzelnen:

- Kodierung
- String-Literale
- String-Interpolation
- Objekte und Methoden

Umlaute & andere Sorgenkinder

- Aus Sicht des Computers bestehen Python-Programme (wie alle Dateien) aus einer Folge von *Bytes*.
- Aus unserer Sicht bestehen sie aus einer Folge von *Zeichen*.
- Um die Sichten zu verbinden, verwendet der Computer eine Abbildung von Zeichen auf Bytes (*Kodierung*).
  - Leider ist diese Kodierung bei verschiedenen Betriebssystemen unterschiedlich: Nur Byte-Werte im Bereich 0–127 haben eine (einigermaßen) standardisierte Interpretation (ASCII).
  - Bei manchen Kodierungen werden bestimmte Zeichen auch durch mehrere Bytes kodiert (Beispiel: Nicht-ASCII-Zeichen in UTF-8), bei anderen sogar alle (Beispiel: UTF-16).

Kodierungs-Spezifikationen

- Damit Python-Programme plattformunabhängig funktionieren können, sollten sie daher angeben, unter welcher Kodierung sie erstellt wurden. Der Default ist UTF-8.
- Diese Angabe geschieht mit einem speziellen Kommentar, der in der ersten oder zweiten Zeile des Programms stehen muss, falls die Kodierung vom Default abweicht:

umlaute.py

```
# -*- coding: utf-8 -*-
print("äöü")
```

- UTF-8-Dateien benötigen keine Kodierungs-Spezifikationen.
- Gute Python-Editoren (z.B. Emacs) erkennen solche Kodierungs-Deklarationen auch automatisch und verwenden dann die dort angegebene Kodierung.

Mehr zu Strings & ein paar Worte zu Objekten

Wir befassen uns mit den tausend Möglichkeiten, einen String zu notieren, erzeugen vernünftige Ausgaben und streifen den Begriff des Objekts.

Im Einzelnen:

- Kodierung
- String-Literale
- String-Interpolation
- Objekte und Methoden

String-Literale

String-Literale können in Python auf viele verschiedene Weisen angegeben werden:

- `"in doppelten Anführungszeichen"`
- `'in einfachen Anführungszeichen'`
- `"""in drei doppelten Anführungszeichen"""`
- `'''in drei einfachen Anführungszeichen'''`
- Jede dieser Varianten mit vorgestelltem „`r`“, also z.B. `r"in doppelten Anführungszeichen mit r"`.

Einfach und dreifach begrenzte Strings

- Die `"doppelte"` Variante verhält sich genau so, wie man es aus C und Java kennt. Man schreibt also zum Beispiel:
  - Newlines als `\n`
  - Backslashes als `\\`
  - doppelte Anführungszeichen als `\"`
- Bei `'einfachen'` Strings muss man doppelte Anführungszeichen nicht mit Backslash schützen (dafür aber einfache).
- Bei `"""solchen"""` und `'''solchen'''` Strings kann man beide Sorten Anführungszeichen sorglos verwenden, sofern sie nicht dreifach auftreten. Außerdem dürfen solche Strings über mehrere Zeilen gehen; die Zeilenenden bleiben wörtlich erhalten.

Beispiele für einfach und dreifach begrenzte Strings

strings.py

```
print("Eine Zeile")
# Eine Zeile
print("Zwei\nZeilen")
# Zwei
# Zeilen
print("Mit Apo'stroph")
# Mit Apo'stroph
print('Mit "Anführungszeichen"')
# Mit "Anführungszeichen"
print("""Über mehrere Zeilen mit "solchen"
und 'solchen' Anführungszeichen.""")
# Über mehrere Zeilen mit "solchen"
# und 'solchen' Anführungszeichen.
```


Rohe Strings

Der r-Präfix kennzeichnet einen *rohen* (raw) String.
Rohe Strings gehorchen etwas komplizierteren Regeln:

- Die Regeln für die *Begrenzung* eines rohen Strings sind genauso wie bei normalen Strings: So sind z.B. `r"di\es\ner hie\"r"` und `r'''Die\\ser\\hi''er'''` zwei rohe Strings.
- Der *Inhalt* eines rohen Strings wird jedoch anders behandelt: In ihm finden keinerlei Backslash-Ersetzungen statt:

Python-Interpreter

```
>>> print(r"di\es\ner hie\"r")
di\es\ner hie\"r
>>> print(r'''Die\\ser\\hi''er''')
Die\\ser\\hi''er
```

Rohe Strings sind für Fälle gedacht, in denen man viele (wörtliche) Backslashes benötigt. Wichtigste Anwendung: reguläre Ausdrücke.

Mehr zu Strings & ein paar Worte zu Objekten

Wir befassen uns mit den tausend Möglichkeiten, einen String zu notieren, erzeugen vernünftige Ausgaben und streifen den Begriff des Objekts.

Im Einzelnen:

- Kodierung
- String-Literale
- String-Interpolation
- Objekte und Methoden

String-Interpolation: Beispiele

- *String-Interpolation* ist ein Feature, das mit C's *sprintf* verwandt ist und am einfachsten am Beispiel zu erklären ist:

Python-Interpreter

```
>>> name = "Gambolputty"
>>> greeting = "Hello, Mr %s." % name
>>> print(greeting)
Hello, Mr Gambolputty.
>>> x, y, z = 7, 6, 7 ** 6
>>> print("%d ** %d = %d" % (x, y, z))
7 ** 6 = 117649
```

- Ab Python 2.6 und 3.0 gibt es eine Alternative, die `format`-Methode von Strings.

Python-Interpreter

```
>>> "{} ** {} = {}".format(2,3,8)
'2 ** 3 = 8'
```

Für Details s. `http://www.python.org/dev/peps/pep-3101/`

String-Interpolation: Erklärung

- String-Interpolation wird vorgenommen, wenn der %-Operator auf einen String angewandt wird. Interpolierte Strings tauchen vor allem im Zusammenhang mit der `print`-Funktion auf, können aber überall verwendet werden.
- Bei der String-Interpolation werden Lücken in einem String durch variable Inhalte ersetzt. Die Lücken werden mit einem Prozentzeichen eingeleitet; zur genauen Syntax kommen wir noch.
- Bei einem Ausdruck der Form `string % ersetzung` muss entweder...
  - `ersetzung` ein Tupel sein, das genau so viele Elemente enthält wie `string` Lücken, oder
  - `string` genau eine Lücke enthalten, in welchem Fall `ersetzung` nicht als Tupel notiert werden muss (aber kann).
- Soll ein Lückentext ein (wörtliches) Prozentzeichen enthalten, notiert man es als %%.

String-Interpolation: `str` und `repr` (1)

- Am häufigsten verwendet man Lücken mit der Notation %s.
  Dabei wird das ersetzte Element so formatiert, wie wenn es mit `print` ausgegeben würde.
  - %s ist also nicht — wie in C — auf Strings beschränkt, sondern funktioniert auch für Zahlen, Listen etc.
- Ein weiterer universeller Lückentyp ist %r.
  Hier wird das ersetzte Element so formatiert, wie wenn es als nackter Ausdruck im Interpreter eingegeben würde.

Diese Buchstaben sind in Analogie zu den builtins `str` und `repr` gewählt, die ihr Argument in der entsprechenden Weise in einen String umwandeln.

String-Interpolation: `str` und `repr` (2)

Python-Interpreter

```
>>> number = 1.0/7.0
>>> print(number)
0.142857142857
>>> print("str: %s  repr: %r" % (number, number))
str: 0.142857142857 repr: 0.14285714285714285
>>> print(str(number))
0.142857142857
>>> print(repr(number))
0.14285714285714285
>>> number
0.14285714285714285
>>> str(number)
'0.142857142857'
>>> repr(number)
'0.14285714285714285'
```


Mindestbreite und Ausrichtung

- Zwischen Lückenzeichen „%“ und Formatierungscode (z.B. `s` oder `r`) kann man eine *Feldbreite* angeben:

Python-Interpreter

```
>>> text = "spam"
>>> print("|%10s|" % text)
|      spam|
>>> print("|%-10s|" % text)
|spam      |
>>> width = -7
>>> print("|%*s|" % (width, text))
|spam    |
```

- Bei positiven Feldbreiten wird rechtsbündig, bei negativen Feldbreiten linksbündig ausgerichtet.
- Bei der Angabe * wird die Feldbreite dem Ersetzungstupel entnommen.

String-Interpolation: Andere Lückentypen

Weitere Lückentypen sind für spezielle Formatierungen spezieller Datentypen gedacht. Die beiden wichtigsten in Kürze:

- `%d` funktioniert für `int`s. Formatierung identisch zu `%s`, aber `%d` wird dennoch häufig verwendet.
- `%f` funktioniert für beliebige (nicht-komplexe) Zahlen. Die Zahl der Nachkommastellen kann mit *.i* oder `.*` angegeben werden. Es wird mathematisch gerundet:

Python-Interpreter

```
>>> zahl = 2.153
>>> print("%f %.1f %.2f" % (zahl, zahl, zahl))
2.153000 2.2 2.15
>>> print("|%*.2f|" % (7, 42))
|  42.00|
>>> print("|%*.*f|" % (10, 3, 3.3 ** 3.3))
|    51.416|
```


String-Interpolation: Anmerkungen

- Ist ein Ersetzungstext zu breit für ein Feld, wird er nicht abgeschnitten, sondern die Breitenangabe wird ignoriert.
- Es gibt noch viele weitere Lückentypen, aber man kommt fast immer mit `%s`, `%r`, `%d` und `%f` aus.
- String-Interpolation wird in Python wegen ihrer Flexibilität sehr häufig eingesetzt — z.B. auch in Situationen, in denen man auch `print` mit mehreren Argumenten verwenden könnte:

Python-Interpreter

```
>>> what = "spam"
>>> amount = 10
>>> print(amount, "pieces of", what)
10 pieces of spam
>>> print("%d pieces of %s" % (amount, what))
10 pieces of spam
```


Mehr zu Strings & ein paar Worte zu Objekten

Wir befassen uns mit den tausend Möglichkeiten, einen String zu notieren, erzeugen vernünftige Ausgaben und streifen den Begriff des Objekts.

Im Einzelnen:

- Kodierung
- String-Literale
- String-Interpolation
- Objekte und Methoden

Objekte und Methoden

- Ich kann es nicht länger verschweigen: Alle Instanzen von Datentypen, die wir bisher gesehen haben (Zahlen, Strings, Listen — sogar Funktionen) sind in Wirklichkeit *Objekte*.
- Damit ist gemeint, dass sie nicht nur aus reinen *Daten* bestehen, sondern auch assoziierte *Attribute* und *Methoden* haben, auf die mit der Punktnotation `ausdruck.attribut` zugegriffen werden kann:

Python-Interpreter

```
>>> x = complex(10, 3)
>>> print(x.real, x.imag)
10.0 3.0
>>> print("spam".index("a"))
2
>>> print((10 + 10).__neg__())
-20
```

Objekte und Variablen

- An dieser Stelle wollen wir eine scheinbar einfache Frage beantworten: Was bewirkt die Anweisung `x = <ausdruck>`?
  - Die naive Antwortet lautet: „Der Variablen `x` wird der Wert `<ausdruck>` zugewiesen.“
  - Eine *bessere*, weil zutreffendere Antwortet, lautet aber eher umgekehrt: „Dem durch `<ausdruck>` bezeichneten Objekt wird der Name `x` zugewiesen.“ Entscheidend ist dabei, dass *dasselbe Objekt* unter *mehreren Namen* bekannt sein kann:

Python-Interpreter

```
>>> food = ["spam", "eggs", "bacon"]
>>> lunch = food
>>> del lunch[0]
>>> print(lunch)
['eggs', 'bacon']
>>> print(food)
['eggs', 'bacon']
```


Fische und Radiosignale (1)

Man stelle sich die Situation so vor:

- Die Daten eines Python-Programms sind Fische (Objekte), die in einem großen Meer schwimmen.
- Einige dieser Fische wurden von Meeresbiologen gekennzeichnet: Sie haben Transponder-Chips (Variablen) in der Haut, über die sie ausfindig gemacht werden könnten.
- Natürlich kann derselbe Fisch mit mehreren Chips (oder auch gar keinem) gekennzeichnet sein.

Fische und Radiosignale (2)

Eine *Zuweisung* wie `x = z + 3` entspricht der Kennzeichnung eines Fischs:

- Zunächst sucht der Meeresbiologe den Fisch mit dem Transponder `z` und holt dann einen neugeborenen Dreierfisch aus einem speziellen Zuchtbecken für Konstanten. Anschließend werden die Fische gepaart und ein Nachkomme ausgesucht.
- Danach überprüft der Meeresbiologe, ob bereits ein Fisch mit dem Transponder `x` im Meer schwimmt. Falls ja, wird er gefangen und wieder ins Meer geworfen, nachdem der Chip entfernt wurde.
- Schließlich wird dem neuen Nachkommen der Chip `x` eingepflanzt, bevor auch er ins Meer geworfen wird.

- Hinweis: Sehr ähnlich wie in Java, nur keine Sonderfälle für primitive Datentypen sowie Typisierung nur von Objekten statt auch von Variablen.

Fische und Radiosignale (3)

Noch mal das Beispiel:

Python-Interpreter

```
>>> food = ["spam", "eggs", "bacon"]
>>> lunch = food
>>> del lunch[0]
>>> print(lunch)
['eggs', 'bacon']
>>> print(food)
['eggs', 'bacon']
```

- Man sagt, dass `food` und `lunch` *dieselbe Identität* aufweisen.

Identität: `is` und `is not`

- Identität lässt sich mit den Operatoren `is` und `is not` testen:
- `x is y` ist `True`, wenn `x` und `y` dasselbe Objekt bezeichnen, und ansonsten `False`.
- `x is not y` ist äquivalent zu `not (x is y)`.

Python-Interpreter

```
>>> x = ["ham", "spam", "jam"]
>>> y = ["ham", "spam", "jam"]
>>> z = y
>>> print(x is y, x is z, y is z)
False False True
>>> print(x is not y, x is not z, y is not z)
True True False
>>> del y[1]
>>> print(x, y, z)
['ham', 'spam', 'jam'] ['ham', 'jam'] ['ham', 'jam']
```


Identität: Die Funktion `id`

- `id(x)` liefert ein `int`, das eine Art „Sozialversicherungsnummer“ für das durch `x` bezeichnete Objekt ist: Zu keinem Zeitpunkt während der Ausführung eines Programms haben zwei Objekte die gleiche `id`.
- `x is y` ist äquivalent zu `id(x) == id(y)`.

Python-Interpreter

```
>>> x = ["ham", "spam", "jam"]
>>> y = ["ham", "spam", "jam"]
>>> z = y
>>> print(id(x), id(y), id(z))
1076928940 1076804076 1076804076
```


Identität: `id`-Recycling

Zu jedem Zeitpunkt haben alle Objekte unterschiedliche `id`s. Es ist allerdings möglich, dass die `id` eines alten Objektes wiederverwendet wird, nachdem es nicht mehr benötigt wird:

recycled-id.py

```
x = [1, 2, 3]
y = [4, 5, 6]
my_id = id(x)
x = [7, 8, 9]
# Das alte Objekt wird nicht mehr benötigt
# => my_id wird frei.
z = [10, 11, 12]
# my_id und id(z) könnten jetzt gleich sein,
# falls Implementierung id wiederverwendet.
```


Identität vs. Gleichheit

- Wir haben es bisher nur bei Strings gesehen, aber man kann Listen und Tupel auch auf Gleichheit testen. Der Unterschied zum Identitätstest ist wichtig:

Python-Interpreter

```
>>> x = ["ham", "spam", "jam"]
>>> y = ["ham", "spam", "jam"]
>>> print(x == y, x is y)
True False
```

- Bei *Gleichheit* wird getestet, ob `x` und `y` den gleichen Typ haben, gleich lang sind und korrespondierende Elemente gleich sind (die Definition ist rekursiv).
- Bei *Identität* wird getestet, ob `x` und `y` dasselbe Objekt bezeichnen.
- Der Gleichheitstest ist verbreiteter; z.B. testet der `in`-Operator (`x in seq`) immer auf Gleichheit, nie auf Identität.

Veränderlich oder unveränderlich?

Jetzt können wir auch genauer sagen, was es mit veränderlichen (*mutable*) und unveränderlichen (*immutable*) Datentypen auf sich hat:

- Instanzen von veränderlichen Datentypen können modifiziert werden. Daher muss man bei Zuweisungen wie `x = y` aufpassen:
  Operationen auf `x` beeinflussen auch `y`.
  - Beispiel: Listen (`list`)
- Instanzen von unveränderlichen Datentypen können nicht modifiziert werden. Daher sind Zuweisungen wie `x = y` völlig unkritisch:
  Da man das durch `x` bezeichnete Objekt nicht verändern kann, besteht keine Gefahr für `y`.
  - Beispiele: Zahlen (`int`, `float`, `complex`), Strings (`str`), Tupel (`tuple`)

Identität von Literalen (1)

- Bei veränderlichen Datentypen wird jedesmal ein neues Objekt erzeugt, wenn ein Literal ausgewertet wird:

mutable-id.py

```
def meine_liste():
  return []
a = []
b = []
c = meine_liste()
d = meine_liste()
# id(a), id(b), id(c) und id(d)
# sind garantiert unterschiedlich.
```


Identität von Literalen (2)

- Bei unveränderlichen Datentypen darf Python ein existierendes Objekt jederzeit „wiederverwenden“, um Speicherplatz zu sparen, muss aber nicht.

immutable-id.py

```
def mein_tupel():
  return ()
a, b, c, d = (), (), mein_tupel(), mein_tupel()
# a, b, c, d eventuell (nicht garantiert!) identisch.

a = 2
b = 2       # a und b sind vielleicht identisch.
c = a       # a und c sind garantiert identisch.
d = 1 + 1   # a und d sind vielleicht identisch.
```

- Wegen dieser Unsicherheit ist es meistens falsch, unveränderliche Objekte mit `is` zu vergleichen.

`x is None`

Eine Anmerkung zu None:

- Die Klasse `NoneType` hat nur eine einzige Instanz (der der Name `None` zugeordnet ist). Daher ist es egal, ob ein Vergleich mit `None` per Gleichheit oder per Identität erfolgt.
- Es hat sich eingebürgert, Vergleiche mit `None` immer als `x is None` bzw. `x is not None` und nicht als `x == None` bzw. `x != None` zu schreiben.
- Der Vergleich per Identität ist auch (geringfügig) effizienter.

Die Operatoren +=, *= & Co.

- Analog zu C, C++ und Java kennt Python die Operatoren
  +=, -=, *=, /=, //=, %=, **=, &=, |=, ^=, <<= und >>=
- Wir haben sie uns bis hierher aufgespart, weil sie sich für veränderliche und unveränderliche Objekte unterschiedlich verhalten:
  - Bei unveränderlichen Objekten ist `x += y` äquivalent zu `x = x + y`.
  - Bei veränderlichen Objekten modifiziert `x += y` das von `x` bezeichnete Objekt; es wird also *kein* neues Objekt erzeugt.

Python-Interpreter

```
>>> a, b = [1, 2], [1, 2]
>>> aa, bb = a, b
>>> a = a + [3, 4]
>>> b += [3, 4]
>>> print(a, aa, b, bb)
[1, 2, 3, 4] [1, 2] [1, 2, 3, 4] [1, 2, 3, 4]
```