

Programmieren in Python

6. Eine kleine Builtin-Safari

Malte Helmert

Albert-Ludwigs-Universität Freiburg

KI-Praktikum, Sommersemester 2009

1 / 34

Eine kleine Builtin-Safari

- ▶ Mittlerweile haben wir ausreichend Konzepte angehäuft, um einen Großteil der Methoden der grundlegenden Datentypen zu verstehen.
- ▶ Daher soll es in dieser Lektion nicht um neue Konzepte gehen, sondern darum, was man mit Strings, Tupeln und Listen so alles anstellen kann.
- ▶ Der Vollständigkeit halber werden einige früher erwähnte Builtins (ohne nähere Beschreibung) wiederholt.

Die Lektion gliedert sich wie folgt:

- ▶ Builtins
- ▶ Methoden von Zahlen und Tupeln
- ▶ Methoden von `list`
- ▶ String-Methoden

2 / 34

Eine kleine Builtin-Safari

- ▶ Mittlerweile haben wir ausreichend Konzepte angehäuft, um einen Großteil der Methoden der grundlegenden Datentypen zu verstehen.
- ▶ Daher soll es in dieser Lektion nicht um neue Konzepte gehen, sondern darum, was man mit Strings, Tupeln und Listen so alles anstellen kann.
- ▶ Der Vollständigkeit halber werden einige früher erwähnte Builtins (ohne nähere Beschreibung) wiederholt.

Die Lektion gliedert sich wie folgt:

- ▶ **Builtins**
- ▶ Methoden von Zahlen und Tupeln
- ▶ Methoden von `list`
- ▶ String-Methoden

3 / 34

Konstruktoren für Zahlen

- ▶ `int(x)`, `long(x)`, `float(x)`, `complex(x)`:
Erzeugt eine neue Zahl des jeweiligen Typs aus einer anderen Zahl **oder einem String** `x`.
- ▶ `complex(re, im)`:
Erzeugt eine komplexe Zahl aus zwei Zahlen `re` und `im`, die Real- und Imaginärteil angeben.

Python-Interpreter

```
>>> x, y = int("21"), long(23.1)
>>> print x, y, x + y
21 23 44
```

4 / 34

Konstruktoren für Strings & Ähnliches

- ▶ `str(x)`, `unicode(x)`:
Formatiert `x` als Byte- bzw. Unicode-String. Die Formatierung ist dieselbe wie bei `print x`.
- ▶ `repr(x)`:
Formatiert `x` als Byte-String. Die Formatierung ist dieselbe wie bei der Ausgabe nackter Ausdrücke im interaktiven Interpreter.
 - ▶ `repr(x)` kann auch als `'x'` (in Backticks) geschrieben werden, dies ist aber verpönt.
- ▶ `chr(number)`, `unichr(number)`:
Erzeugt einen einelementigen Byte- bzw. Unicode-String mit dem Zeichen mit der Kodierung `number`.
- ▶ `ord(char)`:
Nimmt einen einelementigen Byte- oder Unicode-String und liefert die Kodierung des Zeichens.

5 / 34

Konversionen zwischen Zahlensystemen

- ▶ `hex(n)`, `oct(n)`:
Kodiert eine Zahl im Hexadezimal- bzw. Oktalsystem. Liefert Bytestring mit Präfix `0x` bzw. `0` und evtl. Suffix `L`.
- ▶ `int(string, base)`, `long(string, base)`:
Erzeugt eine Zahl aus einer Kodierung im Zahlensystem mit der Basis `base`.
`base = 0` ist ein Spezialfall und versteht genau die gültigen `int`- bzw. `long`-Literale (inkl. den Präfixen `0x` und `0`).

Python-Interpreter

```
>>> print hex(15 ** 15), oct(7 ** 7)
0x613B62C597707EFL 03110367
>>> print int("110101", 2), int("37", 8)
53 31
>>> print int("0xff", 0), int("037", 0), int("45", 0)
255 31 45
```

6 / 34

Konstruktoren für Tupel und Listen

- ▶ `tuple(seq)`:
Erzeugt ein Tupel mit denselben Elementen wie die Sequenz `seq`.
- ▶ `list(seq)`:
Erzeugt eine Liste mit denselben Elementen wie die Sequenz `seq`.
Nützlich zum (flachen) Kopieren von Listen.

Python-Interpreter

```
>>> print list("abc"), tuple(["ham", "spam"])
['a', 'b', 'c'] ('ham', 'spam')
>>> x = [1, [2, 2.5], 3]
>>> y = list(x)
>>> del y[2]
>>> del y[1][1]
>>> print x, y
[1, [2], 3] [1, [2]]
```

7 / 34

Konstruktoren für Bools

- ▶ `bool(x)`:
Erzeugt folgenden `bool`-Wert:
 - ▶ `False`, falls `x` den Wert `False` oder `None` hat, eine Zahl mit Wert `0` (`0`, `0L`, `0.0`, `0j`) oder eine leere Sequenz ist (`""`, `()`, `[]`).
 - ▶ `True` ansonsten.

`bool(x)` wird (implizit) vor Anwendung des `not`-Operators aufgerufen¹; man kann also beispielsweise mit `if not x` testen, ob eine Liste leer ist.

¹Tatsächlich passiert etwas anderes, aber man kann es sich so vorstellen...

8 / 34

Mathematische Funktionen

- ▶ `abs(x)`:
Berechnet den Absolutbetrag der Zahl `x`.
- ▶ `divmod(x, y)`:
Berechnet `(x // y, x % y)`.
- ▶ `pow(x, y[, z])`:
Berechnet `x ** y` bzw. `(x ** y) % z`.
- ▶ `sum(seq)`:
Berechnet die Summe einer Zahlensequenz.
- ▶ `min(seq), min(x, y, ...)`:
Berechnet das Minimum einer Sequenz (erste Form)
bzw. der Argumente (zweite Form).
 - ▶ Sequenzen werden lexikographisch verglichen.
 - ▶ Bei Mischung konzeptuell unvergleichbarer Typen (etwa Zahlen und Listen), ist das Ergebnis willkürlich, aber konsistent.
- ▶ `max(seq), max(x, y, ...)`:
↪ analog zu `min`

9 / 34

Beispiele zu `sum, min, max`

Python-Interpreter

```
>>> primes = (7, 5, 3, 2, 13, 11)
>>> dish = ["ham", "spam", "sausages", "baked beans"]
>>> for seq in (primes, dish):
...     print min(seq), max(seq)
...     print sum(seq)
...
2 11
41
baked beans spam
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

10 / 34

Weitere wichtige Builtins (1)

- ▶ `zip(seq1, ...)`:
Erzeugt Liste von Tupeln von korrespondierenden Elementen der übergebenen Sequenzen.
- ▶ `any(seq)`:
Äquivalent zu `elem1 or elem2 or elem3 or ...`, wobei `elemi` die Elemente von `seq` sind.
Sequenzelemente werden nur bis zum ersten "wahren" Element evaluiert (short-circuit-Semantik).
- ▶ `all(seq)`:
Äquivalent zu `elem1 and elem2 and elem3 and ...`, wobei `elemi` die Elemente von `seq` sind.
Sequenzelemente werden nur bis zum ersten "falschen" Element evaluiert (short-circuit-Semantik).
- ▶ `id(obj)`:
Liefert die Identität eines Objekts (ein `int`).

11 / 34

Weitere wichtige Builtins (2)

- ▶ `len(seq)`:
Berechnet die Länge einer Sequenz.
- ▶ `range([start,] stop[, step])`:
Erzeugt die Liste `[start, start + step, ...]` bis zur Zahl `stop` (exklusive). Bei zwei Argumenten ist `step == 1`, bei einem Argument außerdem `start == 0`.
- ▶ `xrange([start,] stop[, step])`:
Wie `range`, erzeugt aber keine echte Liste.
- ▶ `enumerate(seq)`:
Generiert Paare der Form `(i, seq[i])`
(↪ erzeugt keine echte Liste).

12 / 34

Eine kleine Builtin-Safari

- ▶ Mittlerweile haben wir ausreichend Konzepte angehäuft, um einen Großteil der Methoden der grundlegenden Datentypen zu verstehen.
- ▶ Daher soll es in dieser Lektion nicht um neue Konzepte gehen, sondern darum, was man mit Strings, Tupeln und Listen so alles anstellen kann.
- ▶ Der Vollständigkeit halber werden einige früher erwähnte Builtins (ohne nähere Beschreibung) wiederholt.

Die Lektion gliedert sich wie folgt:

- ▶ Builtins
- ▶ **Methoden von Zahlen und Tupeln**
- ▶ Methoden von `list`
- ▶ String-Methoden

13 / 34

Attribute und Methoden von `int`, `long`, `float`, `complex` und `tuple`

- ▶ Die Zahlenklassen haben keine Attribute und Methoden.²
- ▶ Einzige Ausnahme: Wenn `x` ein `complex` ist, bezeichnet `x.real` den Real- und `x.imag` den Imaginärteil.
- ▶ `tuple` besitzt ebenfalls keine (nicht-magischen) Methoden: Alle Operationen auf Tupeln sind Operatoren oder Builtins.
- ▶ Ausnahmen: Tupel in Python 2.6 haben `index`- und `count`-Methoden entsprechend den gleichnamigen `list`-Methoden. Diese existieren aber noch nicht in Python 2.5, zu dem wir hier kompatibel bleiben wollen.

²Stimmt nicht ganz: Sie haben so genannte *magische Methoden*, die für die Implementation der Operatoren wie `+` `-` `*` `/` aufgerufen werden. Dies sind aber interne Dinge, um die wir uns hier nicht kümmern müssen.

14 / 34

Eine kleine Builtin-Safari

- ▶ Mittlerweile haben wir ausreichend Konzepte angehäuft, um einen Großteil der Methoden der grundlegenden Datentypen zu verstehen.
- ▶ Daher soll es in dieser Lektion nicht um neue Konzepte gehen, sondern darum, was man mit Strings, Tupeln und Listen so alles anstellen kann.
- ▶ Der Vollständigkeit halber werden einige früher erwähnte Builtins (ohne nähere Beschreibung) wiederholt.

Die Lektion gliedert sich wie folgt:

- ▶ Builtins
- ▶ Methoden von Zahlen und Tupeln
- ▶ **Methoden von `list`**
- ▶ String-Methoden

15 / 34

Methoden von `list`: Einfügen und Entfernen

Die folgenden Methoden für Listen modifizieren das betroffene Objekt direkt. Sofern nicht anders angegeben, liefern sie alle `None` zurück.

- ▶ `l.append(element)`:
Hängt ein Element an die Liste an.
Äquivalent zu `l += [element]`.
- ▶ `l.extend(seq)`:
Hängt die Elemente einer Sequenz an die Liste an.
Äquivalent zu `l += seq`.
- ▶ `l.insert(index, element)`: Fügt `element` an Position `index` in die Liste ein.
- ▶ `l.pop()`:
Entfernt das letzte Element und liefert es zurück.
- ▶ `l.pop(index)`:
Entfernt das Element an Position `index` und liefert es zurück.

16 / 34

Methoden von `list`: Suchen

Die folgenden Methoden für Listen modifizieren das betroffene Objekt direkt. Sofern nicht anders angegeben, liefern sie alle `None` zurück.

- ▶ `l.index(value[, start[, stop]])`:
Sucht in der Liste (bzw. in `l[start:stop]`) nach einem Objekt mit Wert `value`. Liefert den Index des ersten Treffers zurück.
Erzeugt einen Fehler, falls kein passendes Element existiert.
- ▶ `l.remove(value)`:
Entfernt das erste Element aus der Liste, das gleich `value` ist.
Entspricht `del l[l.index(value)]`.
- ▶ `l.count(value)`:
Liefert die Zahl der Elemente in der Liste, die gleich `value` sind.

17 / 34

Methoden von `list`: Sortieren und Umdrehen

Die folgenden Methoden verändern alle das betroffene Objekt direkt und liefern `None` zurück.

- ▶ `l.sort()`:
Sortiert die Liste. Der Sortieralgorithmus ist stabil.
 - ▶ Normalerweise wird der übliche Vergleich (mit `<=`, `>=` usw.) als Grundlage zur Sortierung gewählt, aber es ist möglich, darauf Einfluss zu nehmen.
 - ▶ Leider fehlen uns dafür im Moment die syntaktischen Mittel, so dass ich auf später vertrösten muss.
- ▶ `l.reverse()`:
Dreht die Reihenfolge der Liste um; entspricht `l[:] = l[::-1]`.

18 / 34

Sortieren und Umdrehen von Tupeln und Strings

- ▶ Da Tupel und Strings unveränderlich sind, gibt es für sie auch keine mutierenden Methoden zum Sortieren und Umdrehen.
- ▶ Zwei weitere Builtins springen in die Bresche:
 - ▶ `sorted(seq)`:
Liefert eine Liste, die dieselben Elemente hat wie `seq`, aber (stabil) sortiert ist. Es gilt das über `list.sort` Gesagte.
 - ▶ `reversed(seq)`:
Generiert die Elemente von `seq` in umgekehrter Reihenfolge.
Liefert wie `enumerate` einen *Generator* und sollte genauso verwendet werden.

19 / 34

Eine kleine Builtin-Safari

- ▶ Mittlerweile haben wir ausreichend Konzepte angehäuft, um einen Großteil der Methoden der grundlegenden Datentypen zu verstehen.
- ▶ Daher soll es in dieser Lektion nicht um neue Konzepte gehen, sondern darum, was man mit Strings, Tupeln und Listen so alles anstellen kann.
- ▶ Der Vollständigkeit halber werden einige früher erwähnte Builtins (ohne nähere Beschreibung) wiederholt.

Die Lektion gliedert sich wie folgt:

- ▶ Builtins
- ▶ Methoden von Zahlen und Tupeln
- ▶ Methoden von `list`
- ▶ **String-Methoden**

20 / 34

String-Methoden

Wie die meisten Scripting-Sprachen hat Python ein reiches Arsenal an String-Methoden. Wir gliedern in die folgenden Gruppen:

- ▶ Suchen
- ▶ Zählen und Ersetzen
- ▶ Zusammenfügen und Auseinandernehmen
- ▶ Zeichen abtrennen
- ▶ Ausrichten
- ▶ Groß- und Kleinschreibung
- ▶ Zeichentests

Soweit nicht anders erwähnt, werden die Methoden sowohl von `str` als auch von `unicode` unterstützt.

21 / 34

String-Methoden: Suchen (1)

- ▶ `s.index(substring[, start[, stop]])`:
Liefert analog zu `list.index` den Index des ersten Auftretens von `substring` in `s`. Im Gegensatz zu `list.index` kann ein *Teilstring* angegeben werden, nicht nur nach einem einzelnen Element.
- ▶ `s.find(substring[, start[, stop]])`:
Wie `s.index`, erzeugt aber keinen Fehler, falls `substring` nicht in `s` enthalten ist, sondern liefert dann `-1` zurück.
- ▶ `s.rindex(substring[, start[, stop]])`,
`s.rfind(substring[, start[, stop]])`:
Wie `index` bzw. `find`, liefert aber den *letzten* (rechtesten) Treffer.

22 / 34

String-Methoden: Suchen (2)

- ▶ `s.startswith(prefix[, start[, stop]])`:
Liefert `True`, falls `s` (bzw. `s[start:stop]`) mit `prefix` beginnt, sonst `False`.
- ▶ `s.endswith(suffix[, start[, stop]])`:
Liefert `True`, falls `s` (bzw. `s[start:stop]`) mit `suffix` endet, sonst `False`.

Es können für `prefix` und `suffix` auch Tupel von Strings übergeben werden. In diesem Fall wird getestet, ob der String mit *einem* der übergebenen Strings beginnt/endet.

23 / 34

String-Methoden: Zählen und Ersetzen

- ▶ `s.count(substring[, start[, stop]])`:
Berechnet, wie oft `substring` als (nicht-überlappender) Teilstring in `s` enthalten ist.
- ▶ `s.replace(old, new[, count])`:
Ersetzt im Ergebnis überall den Teilstring `old` durch `new` ersetzt. Wird das optionale Argument angegeben, werden maximal `count` Ersetzungen vorgenommen.
Es ist kein Fehler, wenn `old` in `s` seltener oder gar nicht auftritt.

24 / 34

String-Methoden: Zusammenfügen und Auseinandernehmen (1)

- ▶ `s.join(seq)`:
seq muss eine Sequenz (z.B. Liste) von Strings sein.
Berechnet `seq[0] + s + seq[1] + s + ... + s + seq[-1]`.
Häufig verwendet für Komma-Listen:
`", ".join(["ham", "spam", "egg"]) == "ham, spam, egg"`

25 / 34

String-Methoden: Zusammenfügen und Auseinandernehmen (2)

- ▶ `s.split()`:
Liefert eine Liste aller Wörter in `s`, wobei ein „Wort“ ein Teilstring ist, der von Whitespace (Leerzeichen, Tabulatoren, Newlines etc.) umgeben ist.
- ▶ `s.split(separator)`:
Mit der ersten Form identisch, falls `separator` `None` ist.
Ansonsten muss `separator` ein String sein und `s` wird dann an den Stellen, an denen sich `separator` befindet, zerteilt. Es wird die Liste der Teilstücke zurückgeliefert, wobei anders als bei der ersten Variante leere Teilstücke in die Liste aufgenommen werden.

26 / 34

String-Methoden: Zusammenfügen und Auseinandernehmen (3)

- ▶ `s.split(separator, maxsplit)`:
Verhält sich wie `s.split(separator)`, außer wenn das Ergebnis mehr als `maxsplit` Elemente hätte.
 - ▶ In diesem Fall werden nur die ersten `maxsplit - 1` Teilstücke inklusive Separator abgetrennt und das unzerteilte Reststück bildet das letzte Element der Ergebnisliste.
- ▶ `s.rsplit([separator[, maxsplit]])`:
Wie `split`, arbeitet aber von hinten nach vorne.
Der Unterschied ist nur relevant, wenn `maxsplit` verwendet wird.

27 / 34

String-Methoden: Zusammenfügen und Auseinandernehmen (4)

- ▶ `s.splitlines([keepends])`:
Liefert eine Liste der Zeilen in `s`.
Wie `s.split("\n")` mit folgenden Unterschieden:
 - ▶ Wenn `s` mit einem Zeilenende endet, fügt `split` am Ende der Liste einen leeren String an, `splitlines` nicht.
 - ▶ Wenn das optionale Argument `keepends` übergeben wird und wahr ist, werden die Zeilenendezeichen mit in die Ergebnisliste aufgenommen, und zwar am Ende der Zeilen, die sie beenden.

28 / 34

String-Methoden: Zusammenfügen und Auseinandernehmen (5)

- ▶ `s.partition(separator)`:
Liefert ein 3-Tupel (`anfang`, `mitte`, `ende`), so dass `anfang + mitte + ende == s`. Der String wird also in drei Teile geteilt, und zwar nach folgenden Regeln:
 - ▶ Wenn der Separator im String enthalten ist, ist `anfang` das Anfangsstück von `s` bis vor dem ersten Auftreten des Separators, `mitte == separator` und `ende` der Rest des Strings.
 - ▶ Wenn der Separator *nicht* im String enthalten ist, ist `anfang == s`; `mitte` und `ende` sind dann leere Strings.
- ▶ `s.rpartition(separator)`:
Analog zu `partition`, aber vorgehen "von rechts": bei einem Treffer ist `ende` das Endstück nach dem *letzten* Auftreten von `separator`; bei einem Fehlschlag ist `ende == s`.

29 / 34

String-Methoden: Zeichen abtrennen

- ▶ `s.strip()`, `s.lstrip()`, `s.rstrip()`:
Liefert `s` nach Entfernung von Whitespace an den beiden Enden (bzw. am linken bzw. am rechten Rand).
- ▶ `s.strip(chars)`, `s.lstrip(chars)`, `s.rstrip(chars)`:
Wie die erste Variante, trennt aber keine Whitespace-Zeichen ab, sondern alle Zeichen, die in dem String `chars` auftauchen.
 - ▶ Beispiel: `"banana".strip("ba") == "nan"`

30 / 34

String-Methoden: Ausrichten

- ▶ `s.ljust(width[, fillchar])`:
Fügt im Ergebnis rechts Füllzeichen ein, damit der String mindestens die Breite `width` aufweist. Der String wird also linksbündig ausgerichtet.
Wird kein Füllzeichen übergeben, werden Leerzeichen benutzt.
- ▶ `s.rjust(width[, fillchar])`:
Analog zu `ljust`: Richtet den Ergebnisstring rechtsbündig aus.
- ▶ `s.center(width[, fillchar])`:
Analog zu `ljust`: Zentriert den Ergebnisstring.
- ▶ `s.zfill(width)`:
Füllt das Ergebnis von links mit Nullen auf.
Äquivalent zu `s.rjust(width, "0")`.

31 / 34

String-Methoden: Groß- und Kleinschreibung

- ▶ `s.lower()`:
Ersetzt im Ergebnis alle Groß- durch Kleinbuchstaben.
- ▶ `s.upper()`:
Ersetzt im Ergebnis alle Klein- durch Großbuchstaben.
- ▶ `s.swapcase()`:
Vertauscht im Ergebnis Groß- und Kleinbuchstaben.
- ▶ `s.capitalize()`:
Wie `s.lower()`, aber *das erste Zeichen* wird groß geschrieben.
- ▶ `s.title()`:
Wie `s.lower()`, aber *jeder Wortanfang* wird groß geschrieben. Ein Wortanfang ist ein Zeichen, dem kein Buchstabe vorausgeht.

Achtung: Bytestrings betrachte nur A-Z als Buchstaben. Umlaute bleiben also unberührt.

Unicode-Strings behandeln nationale Zeichen wie Umlaute korrekt.

32 / 34

String-Methoden: Zeichentests

Alle Funktionen auf dieser Folie liefern `True` oder `False` zurück.
Bytestrings leiden auch hier unter Umlautblindheit.

- ▶ `s.isalpha()`, `s.isdigit()`, `s.isalnum()`, `s.isspace()`:
Testet, ob `s` nicht-leer ist und nur aus
Buchstaben/Ziffern/Buchstaben und Ziffern/Whitespace besteht.
- ▶ `s.islower()`:
Testet, ob `s` keine Groß-, aber mind. einen Kleinbuchstaben enthält.
- ▶ `s.isupper()`:
Testet, ob `s` keine Klein-, aber mind. einen Großbuchstaben enthält.
- ▶ `s.istitle()`:
Testet, ob `s` nicht-leer ist und `s == s.title()` gilt.

Ausgelassene String-Methoden

Die folgenden Methoden wurden ausgelassen, weil sie entweder zu
esoterisch oder Unicode-spezifisch sind:

- ▶ `decode`, `encode`
- ▶ `isnumeric`, `isdecimal`
- ▶ `translate`
- ▶ `expandtabs`