Programmieren in Python
6. Eine kleine Builtin-Safari

Robert Mattmüller

Albert-Ludwigs-Universität Freiburg

Handlungsplanungs-Praktikum
Wintersemester 2010/2011

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 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
- Methoden von `list`
- String-Methoden

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 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
- Methoden von `list`
- String-Methoden

Konstruktoren für Zahlen

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

Konstruktoren für Strings & Ähnliches

- `str(x)`:
  Formatiert `x` als String. Die Formatierung ist dieselbe wie bei `print(x)`.
- `repr(x)`:
  Formatiert `x` als String. Die Formatierung ist dieselbe wie bei der Ausgabe nackter Ausdrücke im interaktiven Interpreter.
- `chr(number)`:
  Erzeugt einen einelementingen String mit dem Zeichen mit dem Unicode-Codepoint `number`.
- `ord(char)`:
  Nimmt einen einelementigen Byte- oder Unicode-String und liefert die Kodierung des Zeichens.

Python-Interpreter

```
>>> print(chr(65), chr(1488))
A א
>>> print(ord("A"), ord("א"))
65 1488
```


Konversionen zwischen Zahlensystemen

- `hex(n)`, `oct(n)`, `bin(n)`:
  Kodiert eine Zahl im Hexadezimal-, Oktal- bzw. Binärsystem.
  Liefert String mit Präfix `0x`, `0o` bzw. `0b`.
- `int(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`–Literale (inkl. den Präfixen `0x`, `0o` und `0b`).

Python-Interpreter

```
>>> print(hex(15 ** 15), oct(7 ** 7), bin(5 ** 5))
0x613b62c597707ef 0o3110367 0b110000110101
>>> print(int("10f", 16), int("37", 8), int("110101", 2))
271 31 53
>>> print(int("0xff", 0), int("0o37", 0),
... int("0b10010", 0), int("45", 0))
255 31 18 45
```


Konstruktoren für Tupel und Listen

- `tuple(seq)`:
  Erzeugt ein Tupel mit denselben Elementen wie die Sequenz `seq`.
- `list(seq)`:
  Erzeugt eine Liste mit denselben Elemente 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]]
```


Konstruktoren für Bools

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

  `bool(x)` wird (implizit) vor Anwendung des `not`-Operators aufgerufen[1]; man kann also beispielsweise mit `if not x` testen, ob eine Liste leer ist.

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

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.
  - Der Versuch, das Minimum konzeptuell unvergleichbarer Typen (etwa Zahlen und Listen) zu bilden, führt zu einem `TypeError`.
- `max(seq)`, `max(x, y, ...)`:
  $\rightsquigarrow$ analog zu `min`

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 13
41
baked beans spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```


Weitere wichtige Builtins (1)

- `zip(seq1, ...)`:
  Erzeugt Iterator über Tupel von korrespondierenden Elementen der übergebenen Sequenzen.
- `any(seq)`:
  Äquivalent zu `bool(elem1) or bool(elem2) or bool(elem3) or ...`,
  wobei `elem`*i* die Elemente von `seq` sind.
  Sequenzelemente werden nur bis zum ersten „wahren“ Element evaluiert (short-circuit-Semantik).
- `all(seq)`:
  Äquivalent zu `bool(elem1) and bool(elem2) and bool(elem3) and ...`, wobei `elem`*i* 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`).

Weitere wichtige Builtins (2)

- `len(seq)`:
  Berechnet die Länge einer Sequenz.
- `range([start,] stop[, step])`:
  Erzeugt den Zahlenbereich `start, start + step, ...` bis zur Zahl `stop` (exklusive). Bei zwei Argumenten ist `step == 1`, bei einem Argument außerdem `start == 0`.
- `enumerate(seq)`:
  Erzeugt Iterator über Paare der Form `(i, seq[i])`.

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 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
- Methoden von `list`
- String-Methoden

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

- Zu den wenigen Methoden der Zahlenklassen zählt `bit_length()`, das Anzahl der benötigten Bits zur Binärrepräsentation einer ganzen Zahl angibt und `conjugate()`, das eine (komplexe) Zahl konjugiert.
- Dazu kommen die so genannten *magischen Methoden*, die für die Implementation der Operatoren wie `+ - * /` aufgerufen werden. Dies sind aber interne Dinge, um die wir uns hier nicht kümmern müssen.
- Real- und Imaginärteil einer (komplexen) Zahl `x` erhält man mit `x.real` bzw. `x.imag`.

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 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
- Methoden von `list`
- String-Methoden

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]`, aber effizienter.
- `l.extend(seq)`:
  Hängt die Elemente einer Sequenz an die Liste an.
  Äquivalent zu `l += seq`.
- `l.insert(index, element)`: Fügt `element` vor 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.

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 eine Ausnahme, 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)]`, inkl. evtl. Ausnahmen.
- `l.count(value)`:
  Liefert die Zahl der Elemente in der Liste, die gleich `value` sind.

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]`.

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 *Iterator* und sollte genauso verwendet werden.

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 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
- Methoden von `list`
- String-Methoden

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

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 ein einzelnes Element.
- `s.find(substring[, start[, stop]])`:
  Wie `s.index`, erzeugt aber keine Ausnahme, 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.

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.

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`.
  Wird das optionale Argument angegeben, werden maximal `count` Ersetzungen vorgenommen.
  Es ist kein Fehler, wenn `old` in `s` seltener oder gar nicht auftritt.

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]`, aber viel effizienter.
  Häufig verwendet für
  - Komma-Listen:
    `", ".join(["ham", "spam", "egg"]) == "ham, spam, egg"`
  - Verketten vieler Strings:
    `"".join(list_with_many_strings)`

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.

Python-Interpreter

```
>>> " 1 2 3 ".split()
['1', '2', '3']
>>> "1,,2".split(",")
['1', '', '2']
```


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.

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.
  - Windows-Zeilenenden `\r\n` werden richtig behandelt.

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`.

String-Methoden: Zusammenfügen und Auseinandernehmen (6)

partition.py

```
address = "uni-freiburg.de:8080"
host, match, port = address.partition(":")
if match:
    port = int(port)
else:
    port = 80

print("host: %s, port: %s" % (host,port))
# Ausgabe ist 'host: uni-freiburg.de, port: 8080'
```


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"`

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")`.

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: Umlaute werden zwar korrekt behandelt, ß bleibt aber bei Klein- und Großschreibung immer gleich.

String-Methoden: Zeichentests

Alle Funktionen auf dieser Folie liefern `True` oder `False` zurück.

- `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.