Einführung in das Programmieren

Listen in python

Christoph Draxler, Vanessa Reichel
draxler@phonetik.uni-muenchen.de
vanessa.reichel@campus.lmu.de

10. Mai 2023

Motivation

Viele Anwendungen basieren auf Sammlungen ähnlicher Werte...

- Einkaufslisten
- Wörter eines Satzes
- Temperaturmessungen,...

Datenstrukturen in python

In python gibt es vier Datenstrukturen, die für diese Aufgabe vorgesehen sind:

- `List`,
- `Tuple`, liste fester Elemente, man kann nicht löschen
- `Set` und
- `Dictionary`. eintrag ist der Schlüssel, Wert der zu dieser Schlüssel gehört

Ihnen ist gemein, dass in einer Variablen mehrere Werte gespeichert werden können. Sie unterscheiden sich aber u. a. im Zugriff auf die einzelnen Elemente.

Listen in python

In python erzeugt man eine Liste üblicherweise durch Aufzählung innerhalb einer eckigen Klammer `[]`. Auch die Verwendung des Konstruktors `list()` ist möglich[1].

```
fach = ["Phonetik", "Computerlinguistik", "Anglistik"]
```

oder

```
obst = list(("Apfel", "Birne", "Orange"))
```

Elemente einer Liste können unterschiedliche Datentypen sein, z.B.

```
abba = ["Agnetha", 1951, "Björn", 1945, "Benny", 1946,
  "Anni-Frid", 1945]
```

[1] hier steht das Argument dann in eigenen runden Klammern

Zugriff auf Elemente

Der Zugriff auf ein Element einer Liste erfolgt über seinen *Index*. Das erste Element hat den Index 0, das letzte in einer Liste mit n Elementen den Index $n - 1$.
Der Index steht in eckigen Klammern `[]`.

```
abba[2]  # 3. Element
abba[-2] # das vorletzte Element
```
[-2] fangt von hinten an

Ein positiver Index zählt von vorne, ein negativer von hinten. Die Länge einer Liste berechnet die Funktion `len()`. Die leere Liste `[]` hat die Länge 0.

Auswahl einer Teilliste

Mit einer Bereichsgaben `[n:m]` kann man eine Teilliste der Elemente ab Index n bis $m-1$ auswählen:

```
abba[0:4]  # das 1. bis 4. Element
abba[1:-2] # das 2. bis zum vorvorletzten Element
```

Ist der linke Index leer, wird ab dem ersten Listenelement ausgewählt. Ist der rechte Index leer (oder größer als die Anzahl Elemente in der Liste), wird die Liste bis zum letzten Element ausgewählt.

```
abba[1:1000] # alle Elemente ab dem 2.
```

von rechts nach links ab -1

Exkurs: Zeichenketten sind Listen!

In der letzten Stunde gab es eine Aufgabe zum Prüfen, ob ein Element in einer Liste vorkommt. Dort wurde die Schreibweise `kette[0]` für den Zugriff auf das erste Zeichen der Zeichenkette verwendet: Eine Zeichenkette ist somit nichts anderes als eine Liste von Zeichen.

```
kette = "python"
kette[0] = "p"
kette[-1] = "n"
kette[1:3] = "yt"
```

Listenelemente ersetzen

Elemente an bestimmter Position in der Liste können ersetzt werden.

```
abba[1] = 1950 # setzt das 2. Element auf den Wert 1950
abba[2:4] = ["a", "b"] # ersetzt das dritte und vierte
   Elemement der Liste elementweise durch "a", "b"
```

Passen die Bereichsangabe und die Länge der ersetzenden Liste nicht zueinander wird der alte Bereich gelöscht und die neue Liste eingefügt. Dabei verändert sich die Anzahl der Elemente der neuen Liste entsprechend.

Achtung!

```
abba[2:3] = ["a", "b"]
abba[2] = ["x", "y"]
print(abba)
```

ergibt

```
['Agnetha', 1951, ['x', 'y'], 'b', 1945,
 'Benny', 1946, 'Anni-Frid', 1945]
```

- Einmal wird der Bereich von Position 2 bis vor 3 elementweise ersetzt,
- einmal wird nur das Element an Position 2 durch ein listenwertiges Element ersetzt.

In der Praxis wird es immer klar sein, ob eine List elementweise eingefügt werden soll, oder als ein listenwertiges Element.

Listenoperatoren und -funktionen I

`"Björn" in abba` prüft, ob `Björn` in abba vorkommt.

`del abba[2]` entfernt das 3. Element aus der Liste. Wird die Liste ohne eckige Klammern verwendet, wird die ganze Variable gelöscht.

`len(list)` gibt die Länge der Liste zurück

`list.insert(2, "b")` fügt `"b"` an der dritten Stelle in die Liste `list` ein. Dadurch verlängert sich die Liste um ein Element.

`list.append(x)` fügt ein Element `x` an die Liste `list` an.

`list.extend(abba)` fügt eine Liste abba an die Liste `list` an.

bleibt eine Liste

Listenoperatoren und -funktionen II

`list.remove("Björn")` entfernt das Element `Björn` aus der Liste `list`.

`abba.pop(2)` entfernt das 3. Element aus der Liste; wird `pop()` ohne Argument verwendet, wird das letzte Element gelöscht.

`abba.clear()` löscht alle Elemente der Liste, aber die Variable bleibt erhalten.

`abba.sort()` sortiert die Listenelemente nach ihrem Wert.

`abba.reverse()` kehrt die Reihenfolge der Listenelemente um.

`abba.copy()` kopiert eine Liste.

x.copy(abba) = x ist neue Variable mit der Elemente in Abba

Eigenschaften von Listen

- Listenelemente haben einen numerischen Index
- Eine Liste kann Duplikate enthalten
- Elemente einer Liste können entfernt, ersetzt oder gelöscht, neue Elemente hinzugefügt werden

Der Befehl `type(abba)` gibt `<class 'list'>` zurück.

Beispiel

Sie haben beim Stammtisch Schafkopf gespielt und dabei auch etwas gegessen und getrunken. Der Kellner summiert die Beträge

```
zettel = [3.60, 3.90, 8.20, 2.40]

def summe (list):
   if len(list) == 0:
      return(0)
   else:
      return(list[0] + summe (list[1:]))

summe(zettel)
```

Wie teuer war der Abend – und hätten Sie ihn mit Schafkopf finanzieren können?

Auswahl aus Listen

Mit der sog. *list comprehension* kann man in einer einzigen Zeile eine Auswahl von Elementen einer Liste formulieren:

```
woerter = ["morgen", "morgen", "nur", "nicht", "heute"]
kurzeWoerter = [w for w in woerter if len(w) < 5]
```

alle Element wo len kleiner 5 ist

Eine Variable (hier `w`) steht vor einer `for`-Schleife, die nur eine Bedingung enthält. Das Ergebnis ist eine neue Liste, die alle Elemente der alten enthält, die die Bedingung erfüllen.

Listen von Listen

Die Elemente von Listen können selber wieder Listen sein.

```
wetter = [
  "München",
  ["Do", "Fr", "Sa", "So", "Mo", "Di", "Mi"],
  [14, 12, 11, 11, 11, 19, 6]
]
```

`wetter[1][0]` gibt `'Do'` zurück, `wetter[2][0]` den Wert 14. Das funktioniert, ist aber nicht sehr elegant, weil man die Position der Elemente wissen muss.

Beispiel

Wetteronline sagt für die nächsten Tage die folgenden Temperaturen vorher:

```
meineCelsius = [13, 11, 12, 13, 13, 11, 6,
   7, 4, 5, 6, 8, 9, 9]
```

- Berechnen Sie die Maximaltemperatur
- Berechnen Sie die Minimaltemperatur
- Berechnen Sie die Durchschnittstemperatur

Schreiben Sie jeweils eine Funktion (z.B. `maximal()`, `minimal()` o.ä.), die das gewünschte Ergebnis berechnet. Sie können dies rekursiv und elegant formulieren.

Entwicklung der Funktion `minimal()`

Umgangssprachliche Formulierung: gib' das kleinste Element einer Liste zurück!
Vorgehen: Du brauchst zwei Variablen: `x` für das aktuelle Minimum, und `liste` mit den noch zu prüfenden Werten. In jedem Schritt vergleichst Du `x` mit dem ersten Element der Liste, also mit `liste[0]`. Ist `x` größer als `liste[0]`, dann hast Du ein neues kleinstes Element gefunden. Ersetze `x` durch `liste[0]` und suche weiter im Rest der Liste, also `liste[1:]`. Ist `x` kleiner oder gleich `liste[0]`, dann suche weiter im Rest der Liste. Wenn die zu durchsuchende Liste keine Elemente mehr enthält bist Du fertig – `x` ist das kleinste Element.

Entwicklung der Funktion `minimal()` -- Teil 2

Implementation: die Funktion braucht zwei Argumente, `x` und `liste`. Ist die Liste leer, dann kann die Suche nicht weitergehen und `x` ist das gesuchte Ergebnis. Das ist also der Terminalfall. Enthält die Liste noch weitere Elemente, dann gibt es zwei mögliche rekursive Fortsetzungen: entweder mit dem bisherigen `x`, oder einem neuen kleinsten Wert.

Implementation der Funktion `minimal()`

```
def minimal (x, liste):
    if liste == []:
        # Terminalfall
        return x
    elif x > liste[0]:
        # neuer kleinster Wert gefunden, damit weitersuchen
        return minimal (liste[0], liste[1:])
    else:
        # x ist immer noch kleinster Wert, weitersuchen
        return minimal (x, liste[1:])
```

Listen und Rekursion kombiniert! Schön, nicht wahr?

Dictionary

In python speichert ein *dictionary* sog. Schlüssel-Werte-Paare (engl. *key-value pairs*). Die Reihenfolge der Elemente bleibt wie vorgegeben, es können Elemente gelöscht, geändert und eingefügt werden. Wichtig: es sind keine mehrfachen Elemente mit demselben Schlüssel erlaubt!
Ein *dictionary* ist vom Datentyp `<class 'dict'>`
Ein *dictionary* wird durch Angabe der leeren geschweiften Klammer oder mit dem Aufruf `dict()` erzeugt:

```
lexikon = {}
```

Beispiel I

Hier wird ein dictionary tatsächlich für ein Aussprachelexikon verwendet:

```
lexikon = {
  "Frühlingswetter" : "f R y: l I N s v E t 6",
  "heute" : "h OY t @",
  "ist" : "I s t",
  "schönes" : "S 2: n @ s"
}
```

Links vom ':' steht der Schlüssel, er muss eindeutig sein. Rechts steht der dazugehörige Wert. `lexikon["heute"]` hat also den Wert "h OY t @".

Beispiel II

Hier geht es um ein einzelnes Buch:

```
handbook = {
  "title" : "Handbook of the International Phonetic
     Association",
  "printer" : "Cambridge University Press",
  "year" : 1999,
  "ISBN" : "0-521-63751-1",
  "parts" : ["Introduction", "Illustrations",
     "Appendices"]}
```

Die Datentypen der Werte sind verschieden, auch Listen u. ä. sind als Werte erlaubt – siehe `parts`.

dictionary-Funktionen

`len()` Anzahl der Schlüssel-Werte-Paare

`lexikon.keys()` die Schlüssel von `lexikon`

`lexikon.values()` die Werte von `lexikon`

`lexikon.items()` die Einträge von `lexikon` als Liste von Tupeln

Achtung: um die Schlüssel oder Werte eines `dictionary` als Liste zu bekommen muss man das Ergebnis des Funktionsaufrufs noch konvertieren: `list(handbook.keys())` ergibt `['title', 'printer', 'year', 'ISBN', 'parts']`.

Listen und `dictionary` kombinieren

Sehr häufig ist es sinnvoll, Listen und `dictionary` zu kombieren. Damit lassen sich komplexe Datenstrukturen erstellen, bei gleichzeitig einigermaßen natürlichem Zugriff auf die Elemente:

```
wetter = {
    "München" : [14, 12, 11, 11, 11,  9,  6],
    "Berlin"  : [14, 14, 12, 11, 11, 10,  8],
    "Florenz" : [17, 20, 18, 17, 17, 16, 16]
  }

print(wetter["München"][4])
```