Python (Repetition) Lösungen+ Fragen & Antworten

Frage 6.1

Was bedeutet es, wenn mit `L[i][j]` oder `L[i][j][k]` auf eine Liste `L` zugegriffen wird?

- `L[i][j]` gibt das `j`-te Element des `i`-ten Elements der Liste `L` zurück. Dazu muss das `i`-te Element selbst wieder ein Objekt sein, dessen Elemente durch einen Index ausgewählt werden können (z. B. Listen oder Strings).
- `L[i][j][k]` gibt das `k`-te Element des `j`-ten Elements des `i`-ten Elements der Liste `L` zurück. Auch hier müssen die ersten beiden „inneren“ Elemente nummerierbar sein.

```
L = [[3, 4, 1], 2, [5, [7, 0, 9, 8], 6]]
L[0] = ?
L[2][-1] = ?
L[2][1][2] = ? L = [[3, 4, 1], 2, [5, [7, 0, 9, 8], 6]]
L[0] = [3, 4, 1]
L[2][-1] = ?
L[2][1][2] = ? L = [[3, 4, 1], 2, [5, [7, 0, 9, 8], 6]]
L[0] = [3, 4, 1]
L[2][-1] = 6
L[2][1][2] = ? L = [[3, 4, 1], 2, [5, [7, 0, 9, 8], 6]]
L[0] = [3, 4, 1]
L[2][-1] = 6
L[2][1][2] = 9 L = [[3, 4, 1], 2, [5, [7, 0, 9, 8], 6]]
L[0] = [3, 4, 1]
L[2][-1] = 6
L[2][1][2] = 9
```

Frage 6.2

Welches sind die Regeln zur Bildung von Slices (Teillisten)?

`L = [0, 1, 2, 3, 4, 5, 6, 7]`

(A) Step `s = 1`: `L[i:j:1] = L[i:j]` mit `i < j`

- `L[1:5] = [1, 2, 3, 4]`
- `L[5:1] = []`
- `L[3:100] = [4, 5, 6, 7]`

- `L[4:] = [4, 5, 6, 7]`
- `L[:3] = [0, 1, 2]`
- `L[:] = [0, 1, 2, 3, 4, 5, 6, 7]`
- `L[:-1] = [0, 1, 2, 3, 4, 5, 6]`

(B) Step `s > 0`: `L[i:j:s]` mit `i < j`

- `L[1:5:2] = [1, 3]`
- `L[::3] = [0, 3, 6]`

(C) Step `s < 0`: `L[j:i:s]` mit `j > i`

- `L[4:1:-1] = [4, 3, 2]`
- `L[1:4:-1] = []`
- `L[::-3] = [7, 4, 1]`

Frage 6.3

Wann ist der Wert eines Python-Ausdrucks eine Liste?

1. Bei jeder Slice-Operation:
   - `[3,8,2,9,4][::2] => [3,2,4]`
   - `[3,8,2,9,4][3:4] => [9]`
   - `[3,8,2,9,4][3:3] => []`
2. Bei Operatoren, Funktionen und Methoden, die aufgrund ihrer Semantik eine Liste zurückgeben. Wir kennen:
   - `sorted([5, 2, 7, 1]) => [1, 2, 5, 7]`
   - `2 * [3, 4] + [5] => [3, 4, 3, 4, 5]`
3. Bei List-Comprehensions:
   `[i**2 for i in range(2, 5)] => [4, 9, 16]`

Frage 6.4

Worin besteht der Unterschied zwischen `L.pop()` und `L.pop(i)`?

- Ist `L` eine Liste, gibt `L.pop()` das letzte Element von `L` zurück und entfernt es aus `L`.

```
A = [3, 5, 7, 9]
x = A.pop()
print(x)          # => 9
print(A)          # => [3, 5, 7]
```

- Ist `L` eine Liste, gibt `L.pop(i)` das `i`-te Element von `L` zurück und entfernt es aus `L`.

  ```
  A = [3, 5, 7, 9]
  x = A.pop(2)
  print(x)          # => 7
  print(A)          # => [3, 5, 9]
  ```

`L.pop()` ist dasselbe wie `L.pop(-1)`.

Frage 6.5

Worin besteht der Unterschied zwischen den Listen-Methoden `L.pop(3)` und `L.remove(3)`?

- `L.pop(3)` entfernt das Element an der Position `3` und gibt es als Wert zurück.

  ```
  A = [7, 3, 6, 4, 5, 3]
  x = A.pop(3)
  print(x)                # => 4
  print(A)                # => [7, 3, 6, 5, 3]
  ```

- `L.remove(3)` entfernt das erste Vorkommen der Zahl `3` und gibt keinen Wert, d. h. `None` zurück.

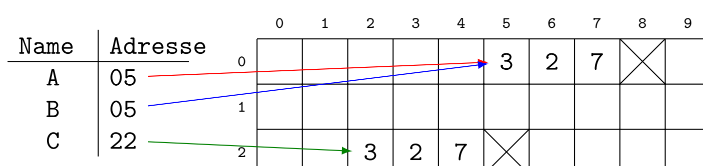
  ```
  A = [7, 3, 6, 4, 5, 3]
  x = A.remove(3)
  print(x)                # => None
  print(A)                # => [7, 6, 4, 5, 3]
  ```

Frage 6.6

Wann teilen sich zwei Variablen eine Liste?

- Weisen wir der Variablen `A` eine Liste zu, wird (stark vereinfacht) eine Referenz auf den Anfang der Liste gespeichert.
- Die Zuweisung `B = A` erzeugt *eine Kopie der Referenz* auf `A`.
- Die Zuweisung `C = A[:]` *kopiert alle Elemente* von `A` und erzeugt eine neue Referenz darauf.

```
A = [3, 2, 7]
B = A
C = A[:]
```



Frage 6.7

Was ist in Python eine *List-Comprehension*?

Eine *List-Comprehension* ist ein syntaktisches Konstrukt in Python, mit der die Elemente einer Liste direkt aus den Elementen eines `range()`-Objekts oder einer bereits existierenden Liste erzeugt werden können.

- `[i**2 for i in range(1, 10, 2)]`

  ⇒ `[1, 9, 25, 49, 81]`

- `[10*x for x in [3, 8, 5, 1]]`

  ⇒ `[30, 80, 50, 10]`

Frage 7.1

Welches sind die drei Hauptgründe für die Verwendung von Funktionen?

- Funktionen vermeiden Code-Wiederholungen. Das bedeutet, dass wir nur eine Kopie des Codes einer Operation haben die in Zukunft aktualisiert werden muss.
- Funktionen ermöglichen die Wiederverwendung von Code. Dies macht den in Funktionen eingebetteten Code zu einem wiederverwendbaren Werkzeug, das in einer Vielzahl von Programmen aufgerufen werden kann.
- Funktionen ermöglichen die Aufteilung eines komplexen Systems in überschaubare Teile, wobei jeder dieser Teile einzeln entwickelt werden kann.

Frage 7.2

Zu welchem Zeitpunkt wird in Python eine Funktion erzeugt?

Eine Funktion wird dann erzeugt, wenn der Python-Interpreter aus dem Code in der `def`-Anweisung ein Funktionsobjekt erzeugt und es dem Namen der Funktion zugewiesen hat.

```
def rechteck_umfang(a, b):
    return 2*(a + b)
```

Frage 7.3

Wann wird der Code ausgeführt, der innerhalb einer Funktionsdefinition steht?

Der Funktionsrumpf (der Code in einem `def`-Statement) wird jedesmal dann ausgeführt, wenn die Funktion mit ihrem Namen und allfälligen aktuellen Parametern aufgerufen wird.

```
u = rechteck_umfang(3, 5)
print(u) # => 16
```

Frage 7.4

Welche Möglichkeiten gibt es, beim Aufruf eine Funktion die aktuellen Parameter den formalen Parametern zuzuweisen?

```
def kapitalwert(k, p, t):
    return k*(1 + p/100)**t
```

- Durch die Position der Parameter: Die Reihenfolge der aktuellen Parameter im Funktionsaufruf stimmt mit der in der Funktionsdefinition überein.

  ```
  kapitalwert(2000, 1.5, 10)
  ```

- Durch Schlüsselwort-Parameter: Die aktuellen Parameter werden in beliebigen Reihenfolge den formalen Parametern zugewiesen.

  ```
  kapitalwert(p=1.5, t=10, k=2000)
  ```

`kapitalwert(k, p, t)` berechnet den Wert eines Startkapitals der Höhe `k`, das `t` Jahre zu einem Zinsfuss von `p`% verzinst wird.

Frage 7.5

Wie kann eine Funktion ein von ihr berechnetes Resultat an die aufrufende Instanz übermitteln?

Indem in der Definition der Funktion das Endergebnis unmittelbar nach der `return`-Anweisung steht.

```
def rechteck_umfang(a, b):
    u = 2*(a + b)
    return u

print(10 * rechteck_umfang(2, 5))  => 140
```

Frage 7.6

Was gibt eine Funktion zurück, die keine `return`-Anweisung in ihrer Definition enthält?

Sie gibt `None` zurück.

Frage 7.7

Was bedeutet es, wenn bei der Definition einer Funktion die Parameter mit Standardwerten belegt weden?

Dies bedeutet, dass die Werte dieser Parameter beim Aufruf der Funktion fehlen können. In diese Fall werden sie durch die Standardwerte ersetzt. *Beispiel:*

```
def f(x, y=2, z=10):
    return x + y + z

print(f(3, 1))   # => 14
print(f(7))   # => 19
```

Falls `y` oder `z` kein Wert zugewiesen wird, werden sie automatisch durch `2` und `10` ersetzt.

Frage 7.8

Wie lauten die beiden wichtigsten Regeln für die Gültigkeit von Namen innerhalb und ausserhalb von Funktionen?

Namen von Variablen oder Funktionen, die innerhalb einer Funktion definiert werden, ...

- sind nur innerhalb dieser Funktion „sichtbar“. Von ausserhalb der Funktion kann nicht auf diese Namen zugegriffen werden.
- kollidieren nicht mit identischen Namen, die ausserhalb der Funktion definiert werden.

```
x = 3

def f(x):
    x = x + 1
    return x

print(x)       # 3 (globaler Kontext)
print(f(1))    # 2 (lokaler Kontext)
print(x)       # 3 (globaler Kontext)
```

Frage 7.9

Wie verfährt der Python-Interpreter mit Variablen, die innerhalb einer Funktion stehen?

Wurde der Variablen *innerhalb* der Funktion ein Wert *zugewiesen*?

- Ja: Verwende diesen Wert.
- Nein: Wurde der Variable *ausserhalb* ein Wert zugewiesen?
  - Ja: Verwende diesen Wert.
  - Nein: Gib einen `NameError` aus.

```
b = 3

def f(x):
    a = 10
    print(a+x)
    print(b+x)
    print(c+x)

f(1)            # => 11
                # => 4
                # => NameError
```

Frage 7.10

Wie berechnet Python den Wert `f(6)` für die folgende rekursiv definierte Funktion?

```
def f(x):
    if x < 3
        return 5
    else:
        return x * f(x - 2)
```

Vor dem Erreichen des Base-Case bauen die rekursiven Aufruf mit den lokalen Parametern einen immer grösseren Term auf. Sobald der Base-Case eintrifft, kann der Term durch das „fehlende Puzzlestück“ aufgelöst, d. h. ausgerechnet werden.

$$\texttt{f(6)} \overset{6<3}{=} \texttt{6 * f(6 - 2) = 6 * f(4)}$$
$$\overset{4<3}{=} \texttt{6 * (4 * f(4 - 2)) = 24 * f(2)}$$
$$\overset{2<3}{=} \texttt{24 * 5 = 120}$$

Frage 7.11

Kann in Python eine Funktion Parameter (Argument) einer anderen Funktion sein?

Funktionen sind auch Objekte mit einem Namen und können daher wie Variablen an eine Funktion übergeben werden. Das folgende Beispiel definiert zwei Hilfsfunktionen und dann eine Funktion `f`, welche eine Funktion `g` auf den Wert `x` anwendet.

```
def h1(x):
    return 2*x

def h2(x):
    return x + 1

def f(g, x):
    return g(x)

print(f(h1, 4)) # => 8
print(f(h2, 4)) # => 5
```