Expertenaufgaben: Schleifen in Python

Zusatzaufgaben für Schnelle – for und while vertiefen, transferieren und knobeln

Diese Aufgaben sind Zusatzaufgaben für besonders schnelle und starke Schüler. Du bearbeitest sie, wenn du das Basis-Arbeitsblatt fertig hast. Sie sind deutlich anspruchsvoller: Du musst Code nachvollziehen, begründen, bewerten und eigene Lösungen in neuen Situationen entwickeln. Schreibe bei Programmieraufgaben vollständigen, lauffähigen Code auf und teste ihn, wenn möglich, am Rechner. Du darfst die Aufgaben in beliebiger Reihenfolge bearbeiten.

1 Was gibt der folgende Code aus?

```
summe = 0
for i in range(10, 0, -3):
    summe = summe + i
print(summe)
```

- ☐ 22
- ☐ 27
- ☐ 18
- ☐ Der Code erzeugt eine Fehlermeldung, da die Schrittweite negativ ist.

2 Trace-Aufgabe (ohne Computer): Was gibt der folgende Code aus? Erstelle eine Tabelle, die die Werte von a, b und der Schleifenvariable in jedem Durchlauf zeigt, und begründe das Ergebnis.

```
a = 0
b = 1
for i in range(6):
    print(a, end=' ')
    a, b = b, a + b
```

3 Collatz-Folge: Starte mit einer natürlichen Zahl n. Ist n gerade, wird n halbiert (n // 2), ist n ungerade, wird 3·n + 1 berechnet. Das wiederholt man, bis n = 1 erreicht ist. Schreibe ein Programm, das für ein beliebiges Startn die Folge ausgibt und zählt, wie viele Schritte nötig waren. Begründe, warum hier eine while-Schleife besser geeignet ist als eine for-Schleife. Gib die Folge für n = 6 an.

4 Primzahltest: Schreibe ein Programm, das prüft, ob eine Zahl z > 1 eine Primzahl ist. Nutze eine for-Schleife und den Modulo-Operator %. Verbessere dein Programm anschließend so, dass die Schleife abbricht (break), sobald ein Teiler gefunden wurde. Erkläre, warum es reicht, nur Teiler bis zur Wurzel von z zu testen.

5 Zahlenraten mit Strategie: Der Computer denkt sich eine Zahl zwischen 1 und 100 aus. Du rätst per Halbierungsverfahren (immer die Mitte des noch möglichen Bereichs). a) Schreibe ein Programm mit while-Schleife, das die Zahl geheim = 73 mit diesem Verfahren findet und die Anzahl der Versuche ausgibt. b) Begründe, wie viele Versuche im ungünstigsten Fall höchstens nötig sind, und erkläre den Zusammenhang mit der Zweierpotenz.

6 Entscheide, ob die Aussagen wahr oder falsch sind. Überlege dir jeweils eine Begründung.

	richtig	falsch
Jede for-Schleife lässt sich durch eine while-Schleife ersetzen.	☐	☐
Jede while-Schleife lässt sich durch eine for-Schleife mit range() ersetzen, ohne dass man die Anzahl der Durchläufe vorher kennt.	☐	☐
Eine break-Anweisung in der inneren von zwei verschachtelten Schleifen beendet auch die äußere Schleife.	☐	☐
Der Code for i in range(3): for j in range(4): print('*') gibt insgesamt 12 Sterne aus.	☐	☐
Wenn man die Schleifenvariable i innerhalb des Schleifenkörpers einer for-Schleife über range(5) ändert (z. B. i = 10), verändert das die Anzahl der Durchläufe.	☐	☐

7 Fehlersuche (Knobelaufgabe): Der folgende Code soll alle Zahlen von 1 bis 20 ausgeben, die durch 3 oder durch 5 teilbar sind, und am Ende deren Summe nennen. Er enthält drei Fehler. Finde sie, erkläre jeweils, was falsch ist, und schreibe den korrigierten Code auf.

```
summe = 0
for i in range(1, 20):
    if i % 3 == 0 and i % 5 == 0:
        print(i)
        summe = i
print("Summe:", summe)
```

8 Muster zeichnen: Schreibe ein Programm mit verschachtelten Schleifen, das für eine Höhe h = 4 folgende Zahlenpyramide ausgibt (jede Zeile ist zentriert, die Zeile k enthält k-mal die Ziffer k):

```
   1
  22
 333
4444
```

Erkläre, wie die Anzahl der Leerzeichen und die Anzahl der Ziffern mit der Zeilennummer zusammenhängen. Erweitere dann gedanklich: Wie viele print-Zeichen (Ziffern) werden insgesamt für beliebiges h ausgegeben? Gib eine Formel an.

9 Bewerten und vergleichen: Zwei Schüler wollen die Summe aller Zahlen von 1 bis n berechnen. Schüler A nutzt eine for-Schleife mit Summenbildung, Schüler B nutzt die Formel n * (n + 1) // 2. a) Schreibe beide Lösungen als Code auf. b) Vergleiche beide Lösungen hinsichtlich Rechenaufwand für n = 1 000 000 und begründe, welche du bevorzugen würdest. c) Nenne ein Beispiel, bei dem man trotzdem eine Schleife benötigt, weil es keine einfache Formel gibt.

10 Ordne jedem Problem die am besten geeignete Lösungsidee zu.

1. Alle Teiler einer Zahl z finden	____	A) for-Schleife mit range(1, z + 1) und Modulo-Test
2. Eine Zahl so oft halbieren, bis sie kleiner als 1 ist, und die Schritte zählen	____	B) zwei verschachtelte for-Schleifen (8 × 8)
3. Alle Paare (Zeile, Spalte) eines Schachbretts durchlaufen	____	C) while-Schleife mit Zähler
4. Passwort-Eingabe wiederholen, bis es stimmt (Anzahl der Versuche unbekannt)	____	D) for-Schleife mit range(1, 21)
5. Die ersten 20 Quadratzahlen ausgeben	____	E) while-Schleife mit Abbruchbedingung

11 Forscheraufgabe: Ein Kapital von 1000 € wird mit 3 % Zinsen pro Jahr verzinst (Zinseszins, keine Auszahlung). Schreibe ein Programm mit while-Schleife, das berechnet, nach wie vielen Jahren sich das Kapital mindestens verdoppelt hat. Prüfe dein Ergebnis mit der Faustregel „70 geteilt durch den Zinssatz in Prozent“ und beurteile, wie gut die Regel passt.

Lösungen

Expertenaufgaben: Schleifen in Python · Informatik · Klasse 9

Für dich als Lehrkraft: leichtere Fassung in einer Minute

QR scannen oder lehreule.de/anpassen/schleifen-in-python-klasse-9 öffnen: kostenloses Konto, dann schreibt Lehreule die leichtere Fassung mit Hilfen — oder Zusatzaufgaben für Schnelle, eine Word-Datei, einen QR-Code für die Klasse.

Aufgabe 1: 22

Aufgabe 2: Ausgabe: 0 1 1 2 3 5

Tabelle (Werte zu Beginn des jeweiligen Durchlaufs, vor der Ausgabe):

```
i=0: a=0, b=1 -> Ausgabe 0, danach a=1, b=1
i=1: a=1, b=1 -> Ausgabe 1, danach a=1, b=2
i=2: a=1, b=2 -> Ausgabe 1, danach a=2, b=3
i=3: a=2, b=3 -> Ausgabe 2, danach a=3, b=5
i=4: a=3, b=5 -> Ausgabe 3, danach a=5, b=8
i=5: a=5, b=8 -> Ausgabe 5, danach a=8, b=13
```

Begründung: Die Zeile a, b = b, a + b weist beide Variablen gleichzeitig zu (rechte Seite wird zuerst mit den alten Werten ausgewertet). So wird a zum alten b und b zur Summe der alten Werte. Es entstehen die ersten sechs Fibonacci-Zahlen. Der Parameter end=' ' sorgt dafür, dass die Ausgaben in einer Zeile stehen.

Aufgabe 3:

```
n = 6
schritte = 0
print(n)
while n != 1:
    if n % 2 == 0:
        n = n // 2
    else:
        n = 3 * n + 1
    schritte = schritte + 1
    print(n)
print("Schritte:", schritte)
```

Folge für n = 6: 6, 3, 10, 5, 16, 8, 4, 2, 1 -> 8 Schritte.

Begründung: Vorab ist nicht bekannt, wie viele Wiederholungen nötig sind (die Länge der Folge hängt vom Startwert ab und lässt sich nicht einfach vorhersagen). Die Schleife soll solange laufen, bis eine Bedingung (n == 1) erfüllt ist. Das ist der typische Einsatzfall der while-Schleife. Hinweis: Bis heute ist unbewiesen, ob die Folge für jede Startzahl bei 1 ankommt (Collatz-Vermutung).

Aufgabe 4:
```
z = 29
ist_prim = True
for t in range(2, int(z ** 0.5) + 1):
    if z % t == 0:
        ist_prim = False
        break
if ist_prim:
    print(z, "ist eine Primzahl")
else:
    print(z, "ist keine Primzahl")
```

Erwartete Ausgabe für z = 29: 29 ist eine Primzahl

Erklärung: Mit break endet die Schleife sofort nach dem ersten gefundenen Teiler, weitere Tests wären überflüssig. Warum bis √z? Wenn z = a · b gilt, dann kann nicht sowohl a als auch b größer als √z sein, denn sonst wäre a · b > z. Also hat jede zusammengesetzte Zahl mindestens einen Teiler t mit 2 ≤ t ≤ √z. Findet man bis dahin keinen, ist z prim. Das spart bei großen Zahlen viele Durchläufe.

Aufgabe 5: a)
```
geheim = 73
unten = 1
oben = 100
versuche = 0
while True:
    tipp = (unten + oben) // 2
    versuche = versuche + 1
    if tipp == geheim:
        break
    elif tipp < geheim:
        unten = tipp + 1
    else:
        oben = tipp - 1
print("Gefunden nach", versuche, "Versuchen")
```

Ablauf für 73: Tipps 50, 75, 62, 68, 71, 73 -> 6 Versuche.

b) Mit jedem Versuch halbiert sich die Anzahl der noch möglichen Zahlen. Mit k Versuchen lassen sich höchstens 2^k − 1 Zahlen sicher unterscheiden (1, 3, 7, 15, 31, 63, 127, ...). Da 2^6 − 1 = 63 < 100 ≤ 127 = 2^7 − 1 ist, braucht man im ungünstigsten Fall 7 Versuche. Allgemein: Versuche ≈ $\log_2$(Anzahl der Zahlen). Das Verfahren ist deshalb extrem effizient, auch bei 1 Million Zahlen reichen etwa 20 Versuche.

Aufgabe 6: Jede for-Schleife lässt sich durch eine while-Schleife ersetzen. → **richtig**
Jede while-Schleife lässt sich durch eine for-Schleife mit range() ersetzen, ohne dass man die Anzahl der Durchläufe vorher kennt. → **falsch**
Eine break-Anweisung in der inneren von zwei verschachtelten Schleifen beendet auch die äußere Schleife. → **falsch**
Der Code for i in range(3): for j in range(4): print('*') gibt insgesamt 12 Sterne aus. → **richtig**
Wenn man die Schleifenvariable i innerhalb des Schleifenkörpers einer for-Schleife über range(5) ändert (z. B. i = 10), verändert das die Anzahl der Durchläufe. → **falsch**

Aufgabe 7: Fehler 1: range(1, 20) endet bei 19; die 20 wird nicht geprüft. Richtig: range(1, 21).
Fehler 2: Die Bedingung mit 'and' verlangt Teilbarkeit durch 3 UND 5 (also durch 15). Gewünscht ist 'oder': i % 3 == 0 or i % 5 == 0.
Fehler 3: summe = i überschreibt die Summe in jedem Durchlauf. Richtig: summe = summe + i (bzw. summe += i).

Korrigierter Code:

```
summe = 0
for i in range(1, 21):
    if i % 3 == 0 or i % 5 == 0:
        print(i)
        summe = summe + i
print("Summe:", summe)
```

Ausgabe: 3, 5, 6, 9, 10, 12, 15, 18, 20 und Summe: 98.

Aufgabe 8:

```
h = 4
for zeile in range(1, h + 1):
    zeichen = ""
    for _ in range(h - zeile):
        zeichen = zeichen + " "
    for _ in range(zeile):
        zeichen = zeichen + str(zeile)
    print(zeichen)
```

(Kurzform: print(" " * (h - zeile) + str(zeile) * zeile))

Zusammenhang: In Zeile k stehen h − k Leerzeichen und k Ziffern; zusammen ergibt das in jeder Zeile genau h Zeichen, sodass die Zeilen rechtsbündig abschließen.

Gesamtzahl der Ziffern: 1 + 2 + 3 + ... + h = h · (h + 1) / 2 (Gaußsche Summenformel). Für h = 4: 10 Ziffern.

Aufgabe 9: a)
Schüler A:

```
n = 1000000
summe = 0
for i in range(1, n + 1):
    summe = summe + i
print(summe)
```

Schüler B:

```
n = 1000000
print(n * (n + 1) // 2)
```

Beide Programme geben 500000500000 aus.

b) Schüler A führt eine Million Additionen durch, der Aufwand wächst proportional zu n. Schüler B braucht unabhängig von n nur eine Multiplikation, eine Addition und eine Division (konstanter Aufwand). Für große n ist B deutlich effizienter, daher würde man B bevorzugen. Die Formel geht auf Gauß zurück: Paart man erste und letzte Zahl, ergeben sich n/2 Paare mit der Summe n + 1. Die Schleife ist dafür leichter zu verstehen und flexibel, wenn sich die Aufgabe leicht ändert.

c) Beispiele: Die Summe aller Primzahlen bis n, die Summe aller Zahlen bis n, die durch 3 oder 5 teilbar sind und deren Quersumme gerade ist, oder die Anzahl der Schritte der Collatz-Folge. Hier gibt es keine einfache geschlossene Formel, man muss die Zahlen einzeln prüfen.

Aufgabe 10: 1 → A, 2 → C, 3 → B, 4 → E, 5 → D

Aufgabe 11:

```
kapital = 1000
ziel = 2 * kapital
jahre = 0
while kapital < ziel:
    kapital = kapital * 1.03
    jahre = jahre + 1
print(jahre, "Jahre, Kapital:", round(kapital, 2))
```

Ergebnis: 24 Jahre (Kapital ca. 2032,79 €). Nach 23 Jahren sind es erst ca. 1973,59 €.

Faustregel: 70 / 3 ≈ 23,3 Jahre. Das Ergebnis der Schleife (24 Jahre) liegt sehr nah dran; die Abweichung entsteht, weil die Regel eine Näherung ist (exakt: ln 2 / ln 1,03 ≈ 23,45 Jahre) und weil die Schleife nur ganze Jahre zählt und aufrundet. Eine while-Schleife ist hier passend, da die Anzahl der Jahre vorher nicht bekannt ist.