# Musterlösungen

## Aufgabe 1

### Aufgabenteil a (4 Punkte)

Gefordert war hier nicht eine Eingabemöglichkeit. Möglich ist auch eine Festlegung auf genau 10 Arrayelemente, was den Einsatz der range()-Funktion obsolet macht. Möglich sind auch **for each** oder **while-Schleifen**.

1a.py

```
# Beispielarray
sum_array = [89,45,5,67,78,97,45,4,8,12]

# Anzahl der Elemente bestimmen, um die Laufweite für eine Schleife zu ermitteln
elements_count = len(sum_array)

# Variable zur Berechnung der Summe
array_sum = 0

# Durch das Array laufen und jedes Element auf array_sum aufaddieren
for i in range (0,elements_count):
    array_sum = array_sum + sum_array[i]

# Ausgabe des Ergebnisses
print(array_sum)
```

### Aufgabenteil b (6 Punkte)

Es sind natürlich alternative Lösungen möglich, da die Aufgabe ungenau gestellt war. So konnte man z.B. die beiden Arrays einfach wild mergen und hinterher sortieren. Ich hätte im Prinzip aber sowas gewollt:

1b.py

```
first_array = [0,2,4,6,8,10]
second_array = [1,3,5,7,9,11]

# Array zur Aufnahme der Elemente der beiden anderen Arrays
merged_array = []

# Anzahl der Elemente in beiden Arrays bestimmen
# 1b läuft sonst nicht
# für 1c bräuchte man das sowieso
```

```
elements_count_first = len(first_array)
elements_count_second = len(second_array)

# Wir prüfen, ob beiden Arrays gleich viele Elemente enthalten
if (elements_count_first == elements_count_second):
    # Wenn ja, hängen wir nacheinander die Elemente vom ersten und
zweiten Array an
    for i in range(0,elements_count_first):
        merged_array.append(first_array[i])
        merged_array.append(second_array[i])
    # Ausgabe des neuen Arrays zur Kontrolle
    print(merged_array)
# Wenn beide Arrays ungleiche Anzahlen von Elementen enthalten, brechen
wir ab
else:
    print("Arrays enthalten nicht gleich viele Elemente. Abbruch.")
```

**Aufgabenteil c (4 Zusatzpunkte)**

Das Hauptproblem ist die Laufweite in der for-Schleife. Beim kleineren Array muss man früher aufhören. Diese Lösung sortiert nur sehr eingeschränkt. Man kann des Array später z.B. noch durch Bubblesort schicken, dann wäre das Ergebnisarray auf jeden Fall sortiert.

1c.py

```
first_array = [0,2,3,6,8,10]
second_array = [1,3,5,7,9,11]

# Array zur Aufnahme der Elemente der beiden anderen Arrays
merged_array = []

# Anzahl der Elemente in beiden Arrays bestimmen
# 1b läuft sonst nicht
# für 1c bräuchte man das sowieso
elements_count_first = len(first_array)
elements_count_second = len(second_array)

# Wir prüfen, ob beiden Arrays gleich viele Elemente enthalten
if (elements_count_first == elements_count_second):
    # Wenn ja, hängen wir nacheinander die Elemente vom ersten und
zweiten Array an
    for i in range(0,elements_count_first):
        merged_array.append(first_array[i])
        merged_array.append(second_array[i])
##########################################################################
########### (bekannt, s.o.)

# Wenn beide Arrays ungleiche Anzahlen von Elementen enthalten, müssen
```

```
wir anders vorgehen
# Sonst laufen wir z.B. in der Schleife über die Anzahl der Elemente
heraus
else:
    # Wir schauen, welches Array das größere ist
    # ... und setzen einen Zeiger entsprechend
    # smaller ist jetzt das kleinere, bigger des größere Array
    # Man kann damit genau so arbeiten wie mit den Ursprungsarrays
    if (elements_count_first > elements_count_second):
        bigger = first_array
        smaller = second_array
    else:
        smaller = first_array
        bigger = second_array
    # die erforderliche Laufweite wird durch das größere Array bestimmt
    for i in range(0,len(bigger)):
        # nur ausführen, wenn es noch Elemente im kleineren Array gibt
        if i < len(smaller):
            # Damit es sortiert bleibt, müssen wir schauen, welches
Array an der Stelle i
            # das jeweils kleinere Element enthält und in dieser
Reihenfolge mergen
            if (smaller[i] < bigger[i] ):
                merged_array.append(smaller[i])
                merged_array.append(bigger[i])
            else:
                merged_array.append(bigger[i])
                merged_array.append(smaller[i])
        # Sonst hängen wir einfach die Elemente des größeren Arrays
hintereinander weg
        else:
            merged_array.append(bigger[i])

# Ausgabe des neuen Arrays zur Kontrolle
print(merged_array)
```

# Aufgabe 2

### Aufgabenteil a (4 Punkte)

Diese Aufgabe ist im Prinzip nur etwas Schreibarbeit. Zur Sicherheit sollte man beim Bauen der Sterne im Programmcode den richtigen Datentyp beachten.

2a.py

```
class Star:
   # Attribute des Stern 1:1 aus Aufgabe umsetzen
```

```
   def __init__(self, name, id, distance, apparentMagnitude, type):
      self.name = name
      self.id = id
      self.distance = distance
      self.apparentMagnitude = apparentMagnitude
      self.type = type

# Vorgegebene Sterne bauen
# Dabei beim Anlegen auf die Datentypen achten (z.B. 2000.0 statt 2000)
star01 = Star("Sirius","TYC 5949-2777-1", 8.6, -1.46, "Main sequence")
star02 = Star("Alpha Centauri","TYC 9007-5849-1", 4.37, -0.27, "Main
sequence")
star03 = Star("Rigel","TYC 5331-1752-1", 860.0, 0.13, "Blue
supergiant")
star04 = Star("Almaaz","TYC 2907-1275-1", 2000.0, 2.92, "Yellow
supergiant")
star05 = Star("Luhmann 16","WISE J1049-5319A", 6.589, 14.94, "Brown
dwarf")
```

**Aufgabenteil b (4 Punkte)**

Die Methoden in der Klasse sind nicht so komplex, gerne vergisst man den self-Parameter. Ich habe hier im Hauptprogramm eine Hilfsmethode geschrieben, den den Vergleich unterschiedlicher Sterne vereinfacht. Das ist kein Muss.

2b.py

```
class Star:
   # Attribute des Stern 1:1 aus Aufgabe umsetzen
   def __init__(self, name, id, distance, apparentMagnitude, type):
      self.name = name
      self.id = id
      self.distance = distance
      self.apparentMagnitude = apparentMagnitude
      self.type = type

   # Es muss der Vergleichstern übergeben werden, auf die eigene
Distanz hat man Zugriff über self
   def isCloserThan(self,star_other):
      if (self.distance < star_other.distance):
          return True
      else:
          return False

   # Eine einfache Division reicht als Rückgabewert
   def getDistanceInPC(self):
      return self.distance/3.26
```

```
# Hilfsmethode, um beliebige Sterne vergleichen zu können
def compareDistance(first_star, second_star):
    if (first_star.isCloserThan(second_star)):
        print(f'{first_star.name} ist näher an der Sonne als
{second_star.name}.')
    else:
        print(f'{second_star.name} ist näher an der Sonne als
{first_star.name}.')

# Mainmethode

# Vorgegebene Sterne bauen
# Dabei beim Anlegen auf die Datentypen achten (z.B. 2000.0 statt 2000)
star01 = Star("Sirius","TYC 5949-2777-1", 8.6, -1.46, "Main sequence")
star02 = Star("Alpha Centauri","TYC 9007-5849-1", 4.37, -0.27, "Main
sequence")
star03 = Star("Rigel","TYC 5331-1752-1", 860.0, 0.13, "Blue
supergiant")
star04 = Star("Almaaz","TYC 2907-1275-1", 2000.0, 2.92, "Yellow
supergiant")
star05 = Star("Luhmann 16","WISE J1049-5319A", 6.589, 14.94, "Brown
dwarf")

# Tests
print(star01.getDistanceInPC())
compareDistance(star01,star02)
```

**Aufgabenteil c (6 Punkte)**

Hier gibt es Punkte für jeden Ansatz. Schwierig zu begreifen ist, dass man eigentlich keinen Konstruktor braucht, da ja nur ein klasseninternes Array verwaltet wird. Diese Lösung hier hat natürlich Probleme, z.B. wird gar nicht geprüft, ob ein Stern überhaupt in der Datenbank existiert, wenn man ihn löscht. Da lässt sich beliebig viel weiterer Hirnschmalz hineinstecken. Die get-Methode sollte wahrscheinlich besser in der Stars-Klasse realisiert werden.

2c.py

```
class Star:
   # Attribute des Stern 1:1 aus Aufgabe umsetzen
   def __init__(self, name, id, distance, apparentMagnitude, type):
      self.name = name
      self.id = id
      self.distance = distance
      self.apparentMagnitude = apparentMagnitude
      self.type = type

   # Es muss der Vergleichstern übergeben werden, auf die eigene
Distanz hat man Zugriff über self
```

```
    def isCloserThan(self,star_other):
        if (self.distance < star_other.distance):
            return True
        else:
            return False

    # Eine einfache Division reicht als Rückgabewert
    def getDistanceInPC(self):
        return self.distance/3.26

class StarsDatabase:
    # Wir brauchen keinen Konstruktor, sondern nur ein Array ("index"),
was wir verwalten
    index = []

    # Neue Sterne einfach an unser Array anhängen
    def add(self,star):
        self.index.append(star)
        return

    # Löschen ist genauso einfach (Achtung: Man müsste vorher prüfen, ob
es den Stern überhaupt in der Datenbank gibt!
    def remove(self,star):
        self.index.remove(star)
        print(f'Stern {star.name} gelöscht.')
        return

    # Die Funktion greift nicht auf die Datenbank zu, sollte damit
besser in der Stars-Klasse realisiert werden
    def get(self, star):
        print("Hier die Daten des Sterns:")
        print(star.name)
        print(star.id)
        print(star.distance)
        print(star.apparentMagnitude)
        print(star.type)
        return

    # Hier einfach die Länge des Index-Arrays zurückgeben
    def size(self):
        print(f'Es gibt {len(self.index)} Sterne in der Datenbank.')
        return

# Mainmethode

# Vorgegebene Sterne bauen
# Dabei beim Anlegen auf die Datentypen achten (z.B. 2000.0 statt 2000)
star01 = Star("Sirius","TYC 5949-2777-1", 8.6, -1.46, "Main sequence")
star02 = Star("Alpha Centauri","TYC 9007-5849-1", 4.37, -0.27, "Main
sequence")
star03 = Star("Rigel","TYC 5331-1752-1", 860.0, 0.13, "Blue
```

```
supergiant")
star04 = Star("Almaaz","TYC 2907-1275-1", 2000.0, 2.92, "Yellow
supergiant")
star05 = Star("Luhmann 16","WISE J1049-5319A", 6.589, 14.94, "Brown
dwarf")

# Tests

# Erzeugen der Datenbank
milchstrasse = StarsDatabase()

# Sterne hinzufügen
milchstrasse.add(star01)
milchstrasse.add(star02)
milchstrasse.add(star03)
milchstrasse.add(star04)
milchstrasse.add(star05)

# Überprüfen des Hinzufügens
milchstrasse.size()

# Stern löschen
milchstrasse.remove(star01)

# Überprüfen des Löschens
milchstrasse.size()

# Daten eines Sterns ausgeben
milchstrasse.get(star04)
```