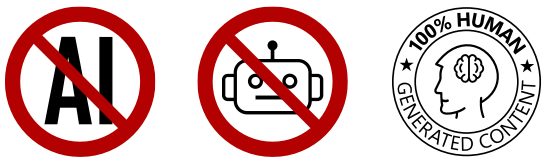
Lösungsalgorithmen

Programmieren Teil 1

Christian Rybovic
L-TIP-24-Do-a



«Python ist eine universelle, üblicherweise interpretierte, höhere Programmiersprache. Sie hat den Anspruch, einen gut lesbaren, knappen Programmierstil zu fördern.»

Inhalt

Hausaufgaben vom 24.04.24 ... 1
Hausaufgaben vom 09.05.24 ... 2
Hausaufgaben vom 16.05.24 ... 3
Hausaufgaben vom 23.05.24 ... 4
Hausaufgaben vom 06.06.24 ... 5
Hausaufgaben vom 13.06.24 ... 6
Hausaufgaben vom 20.06.24 ... 7
Kompetenznachweis 1 vom 04.07.24 ... 8
Kompetenznachweis 2 vom 24.08.24 ... 9
Kompetenznachweis 3 vom 29.08.24 ... 10
Kompetenznachweis 4 vom 05.09.24 ... 11
Kompetenznachweis 5 vom 12.09.24 ... 12
Kompetenznachweis 6 vom 19.09.24 ... 13

Hausaufgaben vom 24.04.24

Im Script bearbeitete Themen:

- Kommentare einzeilig
- Kommentare mehrzeilig
- Print-Funktion einzeilig
- Print-Funktion mehrzeilig (direkt und mit .format())
- Variablen definieren und zuweisen
- Rechenoperationen durchführen

Es wurde ein Python-Script erstellt, welches das Volumen eines Rechtecks anhand der drei Seitenlängen bzw. das Volumen einer Kugel anhand des Radius berechnen kann.

```
"""
Script:   VolumeCalculator.py

Author:   Christian Rybovic
Class:    L-TIP-24-Do-a
Date:     30.04.2024
"""

recSideA = 3   # side A of rectangle in cm
recSideB = 2   # side B of rectangle in cm
recSideC = 5   # side C of rectangle in cm

# output title and rectanlge sides
print("""\
----------------------------
-   Rectangle Calculator   -
----------------------------

Rectangle information:
    Side A = {}cm
    Side B = {}cm
    Side C = {}cm
""".format(recSideA, recSideB, recSideC))

# calculate the volume
recVolume = recSideA * recSideB * recSideC

# output rectangle volume
print("Volume:",str(recVolume) + "cm³")

sphereRadius = 5 # radius of the sphere in cm

# output title and sphere radius
print("""
----------------------------
-    Sphere Calculator     -
----------------------------

Sphere information:
    Radius = {}cm
""".format(recSideA, recSideB, recSideC))

pi = 3.14
sphereVolume = 4/3*pi*sphereRadius*sphereRadius*sphereRadius

# output rectangle volume
print("Volume:",str(sphereVolume) + "cm³")
```

Hausaufgaben vom 09.05.24

Folgende Erweiterungen/Anpassungen wurden am Script vorgenommen:

- Das erste Modul wurde zur Unterstützung importiert (math)
- Für die Berechnung wird nicht mehr fix 3.14 angegeben sondern "math.pi"
- Potenzen werden mit "zahl ** 3" angegeben und nicht mehr 3 Mal multipliziert
- Das Resultat für die Volumen wird auf 2 Nachkommastellen mit "round()" gerundet

```python
import math

def rectangle():
    recSideA = 3   # side A of rectangle in cm
    recSideB = 2   # side B of rectangle in cm
    recSideC = 5   # side C of rectangle in cm

    # output title and rectanlge sides
    print(textRectanlge.format(recSideA, recSideB, recSideC))

    # calculate the volume
    recVolume = recSideA * recSideB * recSideC

    # output rectangle volume
    print("Volume:",str(round(recVolume,2)) + "cm³")

def sphere():
    sphereRadius = 4 # radius of the sphere in cm

    # output title and sphere radius
    print(textSphere.format(sphereRadius))

    sphereVolume = 4/3*math.pi*sphereRadius**3

    # output rectangle volume
    print("Volume:",str(round(sphereVolume, 2)) + "cm³")

rectangle()
sphere()
```

Das Script wurde auch in Funktionen unterteilt. Dadurch ist es übersichtlicher und man kann es besser erweitern. Längere Ausgaben mit "print()" wurden für die Übersichtlichkeit ausserhalb der Funktionen definiert und dort wurden auch grafische Elemente in Form von ASCII-Art eingebaut. Hier als Beispiel das Rechteck:

```
textRectanlge = """\
----------------------------
-   Rectangle Calculator   -
----------------------------
          +------+
         /      /|
        +------+ |
        |      | +
        |      |/
        +------+

Rectangle information:
    Side A = {}cm
    Side B = {}cm
    Side C = {}cm"""
```

Hausaufgaben vom 16.05.24

Am Programm wurden folgende Anpassungen vorgenommen:

- Das Menü wurde eingebaut. Nun kann man auswählen, was man berechnen möchte.
- Es wurden das erste Mal "If-Elif-Else"-Anweisungen verwendet.
- Die Ausgabe der Werte ist neu direkt bei der Grafik, sodass es Visuell besser zur Geltung kommt und man direkt weiss, welcher Wert wo eingesetzt wurde.
- Die Menüauswahl und die Parameter werden neu mit einer dynamischen Benutzereingabe mit "input()" eingelesen und sind nicht mehr statisch im Code hinterlegt.

```
# print menu with available options
print("""\
---------- Menu ----------

[1] Rectangle Calculator
[2] Sphere Calculator

""")

selection = int(input("Select: "))
print("----------------------------")

# call function based on selection
if selection == 1:
    rectangle()
elif selection == 2:
    sphere()
else:
    print("Invalid input")

print("----------------------------")
print("Programm finished...")
```

Hier noch ein Ausschnitt der Eingabe, welche vom Benutzer erfragt wird und direkt in einen Integer umgewandelt wird:

```
def rectangle():
    # request values
    recSideA = int(input("Side A: "))   # side A of rectangle in cm
    recSideB = int(input("Side B: "))   # side B of rectangle in cm
    recSideC = int(input("Side C: "))   # side C of rectangle in cm

    # output title and rectanlge sides
    print(textRectanlge.format(recSideA, recSideB, recSideC))
```

Hausaufgaben vom 23.05.24

Für die Eingabe wurde eine Helper-Funktion mit folgenden Eigenschaften erstellt:

- Die Funktion kann prüfen, ob die Eingabe nur nummerisch ist und keinen Text enthält (mithilfe der Standardfunktion "isnumeric()" von Python).
- Überladen der Funktionen ist nicht ohne weiteres möglich, deshalb wurde für den Parameter "validInput" ein Standard definiert welcher automatisch verwendet wird, wenn nichts eingegeben wurde.
- Die Eingabeabfrage kann nur mit einer gültigen Eingabe unterbrochen werden "while(True)".
- Die Helferfunktion ermöglich eine Vorprüfung der Eingabe, unabhängig des normalen Programmflusses.
- Unschöne Fehlermeldungen werden so mit einer neuen Eingabeabfrage ersetzt.
- Zum ersten Mal wurden if-Anweisungen mit verknüpften Voraussetzungen verwendet (AND und AND NOT).

```python
def requestInput(message, onlyNumeric, validInput = []):
    while(True):
        userInput = input(message)
        if (onlyNumeric and not userInput.isnumeric()):
            continue

        if (validInput and userInput not in validInput):
            continue

        return userInput
```

Ein Aufruf dieser Helper-Funktion schaut im Code dann folgendermassen aus, wobei für die Werte bei der Berechnung einfach auf einen Zahlenwert ohne Text geprüft wird:

```python
# request user input, as long as the condition is not valid
selection = requestInput("Select: ", True, ["1","2"])


recSideA = int(requestInput("Side A: ", True))   # side A of rectangle in cm
recSideB = int(requestInput("Side B: ", True))   # side B of rectangle in cm
recSideC = int(requestInput("Side C: ", True))   # side C of rectangle in cm
```

Für das Menü wurde die if-Anweisung wie von Marco gezeigt in eine elegantere match-Anweisung umprogrammiert:

```python
match selection:
    case "1":
        rectangle()
    case "2":
        sphere()
    case _:
        print("Invalid input")
```

Hausaufgaben vom 06.06.24

Die meisten Programme haben einen Main-Loop, welcher in einer Endlosschleife läuft, bis das Programm beendet wird. In der neuen Version wurde ebenfalls ein Main-Loop implementiert, sodass nach einer Berechnung gleich noch weitere Berechnungen durchgeführt werden können.

Das Menü wird nun anhand eines Dictionary aufgeschlüsselt und angezeigt. Um einen neuen Menüpunkt zu erfassen, muss dieser lediglich dem Dictionary hinzugefügt und der entsprechende Befehl für die Ausführung implementiert werden.

```python
# application main loop
while(True):
    # print menu with available options
    print(textMenu)

    validOptions = []
    for val in shapes:
        validOptions.append(str(list(shapes).index(val)))
        print("[{}] {} Calculator".format(str(list(shapes).index(val)),shapes[val]))

    print("")
    # request user input, as long as the condition is not valid
    selection = requestInput("Select: ", True, validOptions)
    print("----------------------------")

    # call function based on selection
    match selection:
        case "0":
            rectangle()
        case "1":
            sphere()
        case "2":
            cylinder()
        case _:
            print("Invalid input")

    print("----------------------------")
    input("Press Enter to continue...")
    print("----------------------------\n")
```

Auch wurde eine neue Form "Cylinder" eingebaut, welche das Volumen eines Zylinders anhand der Höhe und des Durchmessers berechnen kann.

```python
def cylinder():
    # request value
    cylinderHeight = int(requestInput("Height: ", True)) # height of the cylinderin cm
    cylinderDiameter = int(requestInput("Radius: ", True)) # diameter of the cylinder in cm

    # output title and sphere radius
    print(textCylinder.format(cylinderHeight, cylinderDiameter))

    cylinderVolume = cylinderDiameter*math.pi*cylinderHeight

    # output rectangle volume
    print("Volume:",str(round(cylinderVolume, 2)) + "cm³\n")
```

Hausaufgaben vom 13.06.24

Bei der neuen Version wurde zum ersten Mal mit Dateien gearbeitet. Hierfür kann man die Standardfunktionen "open()", "read/write()" und "close()" verwenden. Die Ausgabe einer Berechnung kann - wenn gewünscht - in eine Datei mit dem Namen "output.txt" gespeichert werden. Die Funktion für die Berechnung des Volumens gibt neu einen String zurück, welcher den Text für die Datei enthält. Die Datei wird im gleichen Ordner erstellt, von wo aus das Script gestartet wurde. Mit folgendem Code wird der Nutzer gefragt, ob er die Berechnung speichern möchte oder nicht:

```
    print("----------------------------")

    # ask if output should be saved to file
    save = requestInput("Do you want to save the output to 'output.txt'? [y,n]: ",
                        False,
                        ["n","y"])
    if save == "y":
        writeToFile()

    print("----------------------------\n")
```

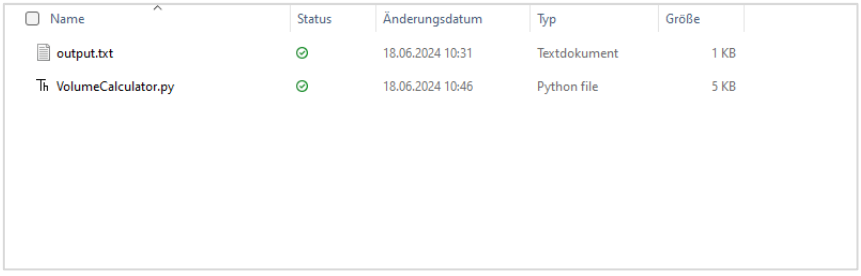
Auch hier wird mithilfe der eigenen Funktion "requestInput()" die Eingabe automatisch auf Gültigkeit geprüft. Gültige Eingaben sind hier nur "y" für Ja und "n" für Nein. Bei "y" wird die folgende Funktion aufgerufen und der Inhalt in die Datei geschrieben.

```
def writeToFile():
    # open file with write access
    f = open("output.txt", "w")

    # write text to file
    f.write(output)

    # close the file
    f.close()

    print("Saved to file 'output.txt'...")
```



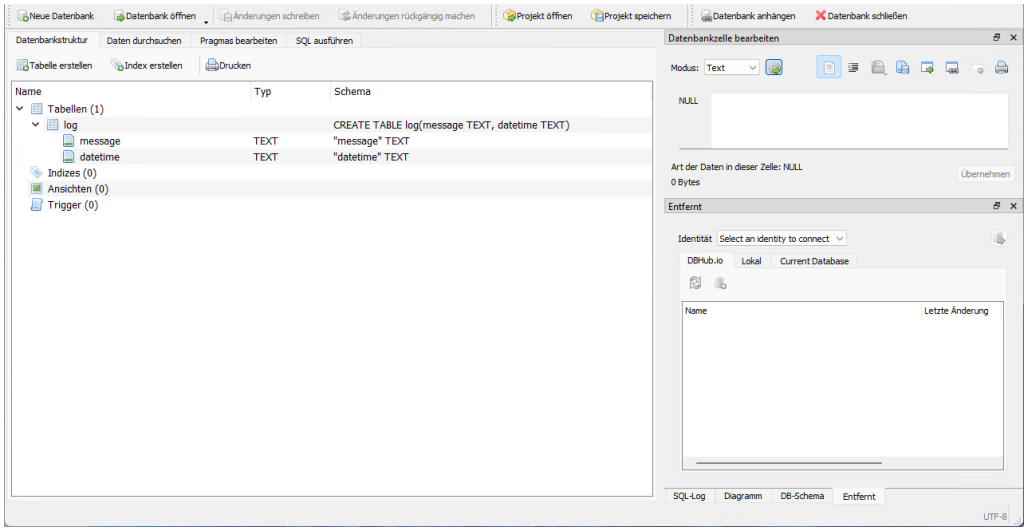
Hausaufgaben vom 20.06.24

Im Programm wurde ein Logging eingebaut, welches Einträge direkt in einer SQLite3-Datenbank erstellt. Zuerst wurden drei Imports und eine Helper-Funktion hinzugefügt. Diese kann im Code an verschiedenen Stellen eingebaut und mit einem Text aufgerufen werden. Dieser Text wird dann in die Datenbank geschrieben. Sollte die Datenbank noch nicht existieren, so wird diese automatisch angelegt:

```python
import sqlite3
import os.path
import datetime

def log(message):
    databaseFile = "log.db"
    if not os.path.isfile(databaseFile):
        # db not exists and needs to be created
        database = sqlite3.connect(databaseFile)
        cursor = database.cursor()
        # primary key omited, auto-generated ROWID
        sql = "CREATE TABLE log(" \
                "message TEXT, " \
                "datetime TEXT)"
        cursor.execute(sql)
        cursor.execute("INSERT INTO log(message, datetime) VALUES (?,?)",
                       (message, str(datetime.datetime.now()),))
        database.commit()
    else:
        database = sqlite3.connect(databaseFile)
        cursor = database.cursor()
        cursor.execute("INSERT INTO log(message, datetime) VALUES (?,?)",
                       (message, str(datetime.datetime.now()),))
        database.commit()
```

Die Datenbank kann anschliessend auch in anderen Programmen (z.B. «DB Browser for SQLite») betrachtet werden. Auf dem folgenden Bild ist die Datenbankstruktur (Tabelle und Felder inkl. Datentyp) ersichtlich. Aktuell wird eine Nachricht und ein Zeitstempel gespeichert:



Kompetenznachweis 1 vom 04.07.24

Mit dem Script werden alle möglichen Kombinationen mithilfe einer **rekursiven Funktion** durchgetestet, bis einer der folgenden zwei Zustände beim Durchlauf eintritt:
- Das Resultat ist weniger 0 → Die Schleife wird abgebrochen
- Das Resultat ist genau 0 → Die Kombination ist gültig und wird als Lösung aufgenommen

```
initialPrice = 480
initialCoins = [200, 100, 50, 20, 10, 5]
combinations = []

def loopCombinations(coins, price, sequence = []):
    while(coins):
        checkCombination(coins, price, sequence)
        coins.pop(0)

def checkCombination(coins, price, sequence):
    newCoins = [] + coins
    newPrice = price - coins[0]
    newSequence = [] + sequence
    newSequence.append(coins[0])

    if(newPrice < 0):
        return

    if(newPrice == 0):
        combinations.append(newSequence)
        return

    loopCombinations(newCoins, newPrice, newSequence)

output = "Price: {0} cents\nCoins: {1}\n".format(initialPrice,
        str(initialCoins).replace('[','').replace(']',''))
loopCombinations(initialCoins, initialPrice)

for index, sequence in enumerate(combinations):
    output += "\nCombination {0}:\n".format(index + 1)
    for coin in dict.fromkeys(sequence):
        output += "{0:>7} x {1:<3} cents\n".format(sequence.count(coin), coin)

output += "\nTotal possible combinations: " + str(len(combinations))
print(output)
with open('output.txt', 'w') as file:
    file.write(output)
```

Es gibt insgesamt **5282 Bezahlkombinationen** für diese Aufgabenstellung.

Da die Ausgabe mit "print()" für jede Zeile sehr langsam ist, wurde zuerst die Ausgabe in einer String-Variable gesammelt und anschliessend der gesamte Inhalt mit einem Befehl ausgegeben. Dadurch konnte die Ausführungszeit von 125 auf 0.13 Sekunden gesenkt werden. Dies entspricht einer Reduktion um 99.896%.

Weiter gilt zu beachten, dass die Thonny Shell für die Ausgabe standardmässig eine Zeilenbeschränkung von 1'000 Zeilen hat. Diese kann auf maximal 100'000 erhöht werden. Auch wenn es für diese Aufgabe ausreichend ist, wird die Ausgabe trotzdem zusätzlich in eine Textdatei geschrieben (output.txt).

Kompetenznachweis 2 vom 24.08.24

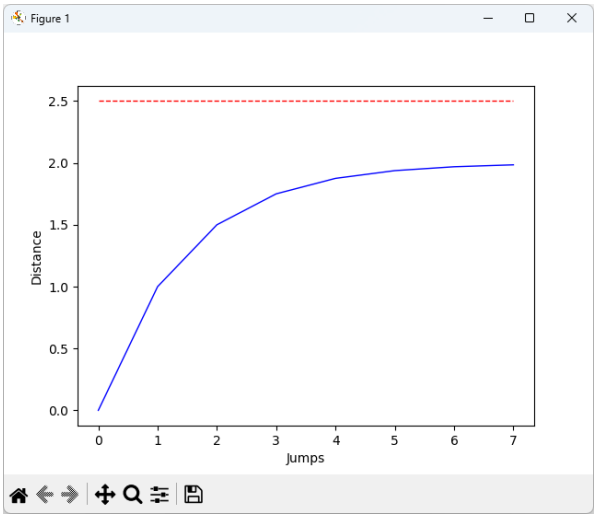
Das Script zeigt auf, ob der Flo/Hase es bis auf die andere Seite der Strasse schafft oder nicht. Mit jedem Sprung halbiert sich die Sprungweite um die Hälfte. Die Anzahl der Sprünge wurde auf 100 limitiert. Dieses Limit genügt für diese Aufgabenstellung, wobei sich bereits nach wenigen Sprüngen eine klare Tendenz zeigt, ob die Strassenüberquerung gelingt oder nicht.

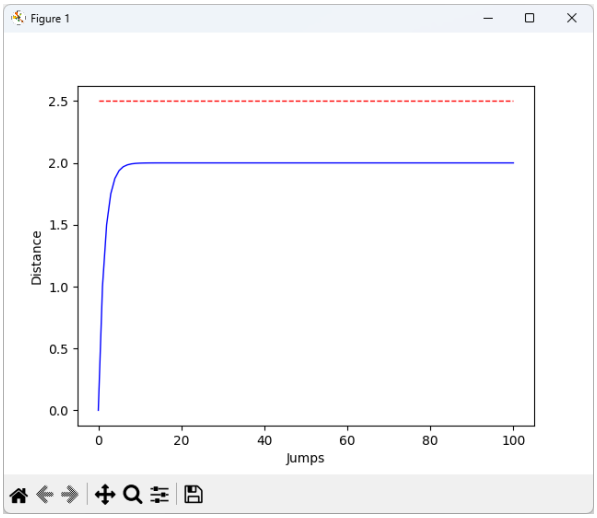
```python
import matplotlib.pyplot as plt

limit, currentStep, stepsBlue, stepsRed = 0, 0, 1, 2.5
valuesBlue, valuesRed = [0], [stepsRed]
plt.show(block=False)

while(limit < 100):
    currentStep += stepsBlue
    stepsBlue = stepsBlue/2
    valuesBlue += [currentStep]
    valuesRed += [stepsRed]
    limit += 1
    plt.clf()
    plt.xlabel("Jumps")
    plt.ylabel("Distance")
    plt.plot(valuesBlue, 'b', linewidth=1)
    plt.plot(valuesRed, 'r', linewidth=1, linestyle='dashed')
    plt.locator_params(axis="x", integer=True)
    plt.pause((stepsBlue+0.01)*3)
```

Für eine bessere und einfachere Darstellung wurde die Python-Bibliothek **matplotlib** verwendet. Bei 2.5 wird eine rote Linie gezeichnet, welche die andere Seite der Strasse anzeigt (das Ziel). Mit Blau wird die gesprungene Weite dargestellt. Mit jedem Sprung wird auch die Grafik schneller aktualisiert.





Das linke Bild zeigt die ersten paar Sprünge, das rechte Bild die Situation nach 100. Hier sieht man deutlich, dass die Distanz auch mit vielen weiteren Sprüngen nicht zu schaffen ist.

Kompetenznachweis 3 vom 29.08.24

In dieser Aufgabe soll berechnet werden, wie viele Reiskörner insgesamt benötigt werden, wenn sich die Anzahl Reiskörner auf jedem Feld eines Schachbretts verdoppelt. Mithilfe einer Formel wäre die Berechnung einfach und schnell gelöst (2^{63}). Die Aufgabenstellung verlangt die Berechnung jedoch ohne Formel.

```
values = [1]
print("Feld {0:02}: {1}".format(1, values[0]))
for num in range(63):
    values += [values[-1]*2]
    print("Feld {0:02}: {1}".format(num+2, values[-1]))

print("\nEs werden total {0:,} Reiskörner benötigt!".format(values[-1]))
```

So berechnet und dargestellt erkennt man das Muster, dass sich alle 3-4 Felder die Gesamtzahl um eine weitere Stelle vergrössert, sprich eine weitere Zehnerpotenz dazu kommt. Somit haben wir hier eine exponentielle Vermehrung der Reiskörner. Bei 64 Schachbrettfeldern benötigt es auf dem letzten Feld **9'223'372'036'854'775'808** Reiskörner.

Gelöst wurde die Aufgabe so, dass jeweils die letzte Zahl im Array mit dem Faktor 2 zum Array hinzugefügt wird. In Python kann man das letzte Element in einem Array mit "**[-1]**" auslesen, was unabhängig der Gesamtanzahl Elementen im Array funktioniert.

Zusätzlich wird die Ausgabe der Felder mit "**{0:02}**" immer zweistellig und die Gesamtanzahl mit "**{0:,}**" für Tausendertrennzeichen angezeigt. Dadurch gibt es eine übersichtlichere Ausgabe, gerade bei so vielen und grossen Zahlen.

```
Feld 49: 281474976710656
Feld 50: 562949953421312
Feld 51: 1125899906842624
Feld 52: 2251799813685248
Feld 53: 4503599627370496
Feld 54: 9007199254740992
Feld 55: 18014398509481984
Feld 56: 36028797018963968
Feld 57: 72057594037927936
Feld 58: 144115188075855872
Feld 59: 288230376151711744
Feld 60: 576460752303423488
Feld 61: 1152921504606846976
Feld 62: 2305843009213693952
Feld 63: 4611686018427387904
Feld 64: 9223372036854775808

Es werden total 9,223,372,036,854,775,808 Reiskörner benötigt!
>>>
```

Kompetenznachweis 4 vom 05.09.24

Im Kapitel 12 wurde gezeigt, wie man mit Python auf Datenbanken zugreifen kann. Um dies zu testen wurde ein Script erstellt, welches beim ersten Start prüft ob es schon eine Datenbank gibt und falls nicht automatisch eine mit Zufallsdaten bereitstellt (products.db).

```
database = sqlite3.connect(databaseFile)
cursor = database.cursor()
# primary key omited, auto-generated ROWID
sql = "CREATE TABLE products(" \
        "name TEXT, " \
        "stock INTEGER, " \
        "price INTEGER, " \
        "created TEXT)"
cursor.execute(sql)
# generate some test-data
cursor.execute("INSERT INTO products(name, stock, price, created) VALUES (?,?,?,?)"
               ("Monitor", 3, 215, str(datetime.datetime.now()),))
cursor.execute("INSERT INTO products(name, stock, price, created) VALUES (?,?,?,?)"
               ("Tastatur", 25, 25, str(datetime.datetime.now()),))
cursor.execute("INSERT INTO products(name, stock, price, created) VALUES (?,?,?,?)"
               ("Maus", 9, 18, str(datetime.datetime.now()),))
cursor.execute("INSERT INTO products(name, stock, price, created) VALUES (?,?,?,?)"
               ("Grafikkarte", 4, 398, str(datetime.datetime.now()),))
cursor.execute("INSERT INTO products(name, stock, price, created) VALUES (?,?,?,?)"
               ("Mainboard", 11, 288, str(datetime.datetime.now()),))
database.commit()
```

Im Script ging es nur darum, mithilfe von Python mit einer Datenbank zu interagieren. Deshalb wurde auf komplexe Codestrukturen und detaillierte Überprüfungen der Eingaben verzichtet. Für ein produktives Script würde man den Code natürlich in diverse Funktionen aufteilen und auslagern.

Aktuell hat man die Möglichkeit, Datensätze hinzuzufügen, zu bearbeiten oder zu löschen. Damit die Änderungen in die Datenbank geschrieben werden, muss das Programm mit dem Befehl "X" beendet werden (nur dann wird "commit" ausgeführt).

```
Key | Name                                | Stock   | Price
-------------------------------------------------------------
  1 | Monitor                             |       3 |     215
  2 | Tastatur                            |      25 |      25
  3 | Maus                                |       9 |      18
  4 | Grafikkarte                         |       4 |     398
  5 | Mainboard                           |      11 |     288
-------------------------------------------------------------

[A] Add new product
[M] Modify product
[D] Delete product

[X] Save and exit

Action:
```

Kompetenznachweis 5 vom 12.09.24

Für diesen Kompetenznachweis wurde ein Script erstellt, welches auf eine MySQL (bzw. um genauer zu sein auf eine MariaDB) Datenbank zugreift. Da ich bereits für Webseiten einen Datenbankserver habe, wurde kurzerhand dort eine zusätzliche Instanz zum Testen erstellt. Mit dem erstellten Python-Script wird über das Internet darauf zugegriffen. Für Python muss vorgängig das Package "**mysql-connector-python**" im Thonny (Windows) installiert werden.

```
print('\n\n♠ ♣ ♥ ♦ ♠ ♣ ♥ ♦ SIMPLE INTERNET CHAT ♠ ♣ ♥ ♦ ♠ ♣ ♥ ♦\n')
cursor.execute(""" SELECT * FROM
                     (SELECT * FROM chat ORDER BY id DESC LIMIT 5)
                   AS subquery ORDER BY id ASC """)
messages = cursor.fetchall()
print('-' * 53)
for message in messages:
    print("{0:>3} | {1:<26}".format(message[0], message[1]))
print('-' * 53)
print('\n[R] Reload messages\n[N] Write new message\n\n[X] Exit\n')
match requestInput("Action: ", ["r", "n", "x"]).lower():
```

Es wurde ein ganz einfacher Chat mit Python realisiert. Beim Starten werden die aktuellsten 5 Nachrichten geladen. Man hat die Möglichkeit mir "R" auf neue Nachrichten zu prüfen oder mit "N" eine Nachricht zu verfassen. Diese wird dann in die Datenbank geschrieben. Für die Abfrage der 5 neusten Nachrichten wurde ein Subquery im SQL erstellt (quasi ein Query in einem Query). Die Datensätze werden auf 5 limitiert und zuerst aufsteigend für die Aktuellsten und dann absteigend für die Anzeige sortiert.

```
SELECT * FROM
  (SELECT * FROM chat ORDER BY id DESC LIMIT 5)
AS subquery ORDER BY id ASC
```

```
♠ ♣ ♥ ♦ ♠ ♣ ♥ ♦ SIMPLE INTERNET CHAT ♠ ♣ ♥ ♦ ♠ ♣ ♥ ♦

-----------------------------------------------------
 10 | Hallo, wie gehts?
 11 | Termin morgen nicht vergessen!
 12 | Einkausliste: Milch, Salami, Brot
 13 | Bitte das Auto noch volltanken...
 14 | Wir sehen uns um 18:00 Uhr
-----------------------------------------------------

[R] Reload messages
[N] Write new message

[X] Exit

Action:
```

Kompetenznachweis 6 vom 19.09.24

Imperativ vs Deklarativ: Bei der imperativen Programmierung gibt man vor, **wie** etwas gemacht werden soll. Bei der deklarativen Programmierung gibt man vor, **was** man haben möchte. Als Beispiel der nachfolgende Code, welcher einfach die Summe ausgeben soll.

```python
numbers = [1, 6, 9, 2, 5]

# Imperativ
result = 0
for number in numbers:
    result += number
print("Total:", result)

# Deklarativ
print("Total:", sum(numbers))
```

Auf beide Arten erhalten wir das gleiche Resultat. Der Weg dahin ist jedoch grundlegend unterschiedlich. Und beide Varianten haben verschiedene Vor- und Nachteile. Man muss immer von Fall zu Fall unterscheiden, welche Variante besser geeignet ist. Nachfolgend eine kurze Gegenüberstellung der Hauptmerkmale:

Imperativ	Deklarativ
+ Flexibler + + Unabhängiger + - Komplizierter - - Zeitintensiver -	+ Schneller + + Einfacher + - Eingeschränkter -

Call-by-Value vs Call-by-Reference: Wie auch in anderen Programmiersprachen kommt es bei Python auf den Datentyp an, welcher einer Funktion übergeben wird. Daraus ergibt sich dann, ob es sich um einen Call-by-Value (also nur der Wert wird übergeben) oder ein Call-by-Reference (eine Referenz zum ursprünglichen Objekt) handelt. **Integers**, **Strings** und **Tupel** sind immer Call-by-Value, das heisst, sie werden innerhalb der aufrufenden Funktion für ausserhalb nicht verändert. Wohingegen z.B. eine **List** per Call-by-Reference an die Funktion übergeben wird. Alle Änderungen innerhalb der Funktion sind dann auch ausserhalb ersichtlich. Falls das nicht gewünscht sein sollte, so kann innerhalb der Funktion zuerst eine Kopie erstellt werden.

Lambda: Gerade im funktionalen Programmieren nehmen Lambda-Ausdrücke einen besonderen Stellenwert ein. Sie sind anonym (also eine Funktion ohne Namen), sie sind keiner Klasse zugeordnet (nicht etwa wie Methoden), können aber wie ein Objekt weitergegeben und bei Bedarf ausgeführt werden. Gerade bei einfachen Operationen können Lambda-Ausdrücke sehr praktisch sein. Sobald sie jedoch komplexer werden, kann der Code schnell unübersichtlich und schlecht lesbar werden.