Kapitel 5: Modularisierung

Unter **Modularisierung versteht** man ein Konzept der Informatik, bei dem Code in Funktionen ausgelagert wird. Damit kann der Code immer wieder verwendet werden, ohne immer wieder erneut aufgeschrieben werden zu müssen.

Außerdem ermöglicht es uns die Modularisierungen, eine Lösung von komplexen Problemen in kleinere, besser beherrschbare, Teilprobleme aufzuteilen. Dieses Prinzip heißt **„Divide and Conquer“**, aus dem Englischen: **„Teile und Herrsche“**.

5.1. Funktionen

Bisher haben wir unsere Programmcode einfach von **oben nach unten** geschrieben und **in dieser Reihenfolge wurde dieser abgearbeitet**.

Jetzt kann es vorkommen, dass wir einige Programmabläufe **öfters benötigen**.
Beispielsweise wollen wir die Uhrzeit am Programmstart und am Programmende ausgeben.

Dazu müssten wir den **gleichen Code** also doppelt - am Anfang und Ende unseres Programmes - schreiben. Das bläht das Programm auf und bringt unnötige Fehlerquellen.

Initialisierung, Aufbau und Aufruf von Funktionen:

Funktionen werden mit Hilfe des `def` -Schlüsselwortes (engl.: Keyword), einem Namen und einer Parameterliste **initialisiert**. Parameterliste bedeutet, dass wir entweder **keinen**, **einen** oder **beliebig viele** Parameter haben können. Betrachten wir dazu einige Beispiele.

In [ ]:
```
def begruessung_ohne_namen():
    print('Hallo! Wie geht es dir? :-)')
```

In [ ]:
```
def begruessung_mit_einem_namen(name):
    print(f'Hallo {name}! Wie geht es dir? :-)')
```

In [ ]:
```
def begruessung_mit_zwei_namen(name1, name2):
    print(f'Hallo {name1} und {name2}! Wie geht es dir? :-)')
```

Was auffallen sollte: Trotz der Verwendung von `print(...)` wird bei der Ausführung der Code-Zeilen **nichts** ausgegeben. Das liegt daran, dass wir die Funktion bisher nur **initialisiert** haben. Damit werden diese noch nicht ausgeführt. Wir müssen die Funktion erst **aufrufen** und die Parameter übergeben (quasi: die "Platzhalter" füllen), damit etwas ausgegeben bzw. ausgeführt wird.

Aufgerufen wird eine Funktion in dem man den Funktionsnamen, gefolgt von runden Klammern, im Code schreibt. Hat man Paramter, so werden diese kommagetrennt in die runden Klammern geschrieben. Auch hierzu nachfolgend die entsprechenden Beispiele.

In [ ]:
```
def begruessung_ohne_namen():
    print('Hallo! Wie geht es dir? :-)')

begruessung_ohne_namen()
```

In [ ]:
```
def begruessung_mit_einem_namen(name):
    print(f'Hallo {name}! Wie geht es dir? :-)')

begruessung_mit_einem_namen('Anna')
begruessung_mit_einem_namen('Benedikt')
begruessung_mit_einem_namen('Clara')
```

In [ ]:
```
def begruessung_mit_zwei_namen(name1, name2):
    print(f'Hallo {name1} und {name2}! Wie geht es dir? :-)')

begruessung_mit_zwei_namen('Anna', 'Benedikt')
begruessung_mit_zwei_namen('Clara', 'David')
begruessung_mit_zwei_namen('Eva', 'Frank')
```

Wie deutlich zu erkennen ist, genügt die **einmalige Initialisierung** der Funktion und wir können diese dann **immer und immer wieder aufrufen**.

Funktionen können **nicht nur** Code direkt ausführen sondern auch einen **Rückgabewert** liefern. Beispielsweise soll eine Funktion eine Berechnung ausführen. Das Ergebnis der Berechnung benötigen wir aber wieder in unserem Programm. Also muss die Funktion diesen Wert **zurückgeben**. Verwendet wird dazu das Schlüsselwort **return**. Alles was hinter `return` steht wird beim **Funktionsaufruf** zurückgegeben und muss dann auch in einer Variable gespeichert werden.

Nachfolgend wird dies am Beispiel eines **Taschenrechners** gezeigt.

5.2. Taschenrechner mit Funktionen

Für einen Taschenrechner benötigen wir mindestens die **vier Grundrechenoperationen**: Addition, Subtraktion, Division und Multiplikation.

Für jede dieser Rechenoperationen definieren wir nun **eine** Funktion. Als **Parameter** werden jeweils zwei Zahlen übergeben und als **Rückgabewert** nutzen wir das Ergebnis der Berechnung.

In [ ]:
```python
def add(a, b):
    ergebnis = a + b
    return ergebnis

def sub(a, b):
    ergebnis = a - b
    return ergebnis

def mul(a, b):
    ergebnis = a * b
    return ergebnis

def div(a, b):
    ergebnis = a / b
    return ergebnis
```

Um die Taschenrechner-Funktionalität zu realisieren implementieren wir nun eine weitere Funktion, welche **alle Funktionen** der Rechenoperationen aufruft. Auch dieser Funktion werden die zwei Zahlen als Paramter übergeben.

In [ ]:
```python
def add(a, b):
    ergebnis = a + b
    return ergebnis

def sub(a, b):
    ergebnis = a - b
    return ergebnis

def mul(a, b):
    ergebnis = a * b
    return ergebnis

def div(a, b):
    ergebnis = a / b
    return ergebnis

def taschenrechner(a, b):
    print(f'Addition: {a} + {b} = {add(a, b)}')
    print(f'Subtraktion: {a} - {b} = {sub(a, b)}')
    print(f'Multiplikation: {a} * {b} = {mul(a, b)}')
    print(f'Division: {a} / {b} = {div(a, b)}')

taschenrechner(10, 5) # VERÄNDERE DIE WERTE! :)
```

An dieser Stelle könnte man nun noch etwas **Feintuning** betreiben: So ist zum Beispiel die Division durch Null nicht definiert und würde zu einem Fehler führen. Dies können wir mit Hilfe einer Verzweigung lösen.

In [ ]:
```python
def add(a, b):
    ergebnis = a + b
    return ergebnis

def sub(a, b):
    ergebnis = a - b
    return ergebnis

def mul(a, b):
    ergebnis = a * b
    return ergebnis

def div(a, b):
    if b == 0:
        return None # verhindert die Division durch Null
    ergebnis = a / b
    return ergebnis

def taschenrechner(a, b):
    print(f'Addition: {a} + {b} = {add(a, b)}')
    print(f'Subtraktion: {a} - {b} = {sub(a, b)}')
    print(f'Multiplikation: {a} * {b} = {mul(a, b)}')
    print(f'Division: {a} / {b} = {div(a, b)}')

taschenrechner(10, 5)
print('---')
taschenrechner(10, 0)
```

Daran wird auch direkt der Nutzen von Funktionen sichtbar. Man hat übersichtliche Einheiten, die leicht zu "beherrschen" sind, und Änderungen müssen nur **ein einziges Mal** an einer einzigen Stelle vorgenommen werden.

5.3. Funktionen

Eine **Programmbibliothek** (in Python als **Module** bezeichnet, auch: libraries) ist eine **Sammlung von Funktionen und Daten**, die in anderen Programmen (wieder-)verwendet werden können. Damit kann man zum einen das Programm "verschlanken", indem wirklich ausschließlich benötigte Funktionalitäten eingebettet werden. Zum anderen sparen sich Programmiererinnen und Programmierer damit unheimlich viel Arbeit. Es gibt "Probleme" die man immer und immer und immer wieder lösen muss.. lagert man die Problemlösung einmalig aus und nutzt diese Lösung dann "nur noch", erspart man sich jedes folgende Mal das komplette neu implementieren.

Um Module zu nutzen, verwendet man das `import-Keyword`. Nachfolgend zwei Beispiele.

Zufallszahlen:

In [ ]:
```
import random

print(random.randint(1, 10))
```

Mit `random.randint(1,10)` wird eine ganze zufällige Zahl zwischen 1 und 10 zurückgegeben.

Mathematische Operationen:

In [ ]:
```python
import math

zahl = math.ceil(5.3)
print('Die Zahl 5.3 aufgerundet ergibt: ', zahl)

zahl = math.floor(5.3)
print('Die Zahl 5.3 abgerundet ergibt: ', zahl)

zahl = math.sqrt(4)
print('Die Quadratwurzel von 4 ergibt: ', zahl)

zahl = math.sin(1)
print('Der Sinus von 1 ergibt: ', zahl)

zahl = math.radians(90)
print('90° im Bogenmaß: ', zahl)

zahl = math.pow(2,4)
print('2^4 ergibt: ', zahl)

zahl = math.exp(5)
print('e^5 ergibt: ', zahl)

zahl = math.sin(math.radians(90))
print('Der Sinus von 90° entpricht: ', zahl)
```

Die Auflistung für beide dieser Bilbiotheken ist **unvollständig**. Weitere Informationen findest du in den jeweiligen **Dokumentationen**:

- math — Mathematical functions
- random — Generate pseudo-random numbers

5.4. Übungen zur Modularisierung

Aufgabe 1 - Potenzfunktion

Implementiere eine Funktion mit dem Namen `potenz`. Als Parameter sollen die zwei ganzen Zahlen `a` und `b` übergeben werden.

Diese beiden ganzen Zahlen müssen **vor** dem Funktionsaufruf durch den Nutzer eingegeben werden.

Die Funktion berechnet die Potenz a^b. Dabei fungiert `a` als Basis und `e` als Exponent. Die Funktion soll das Ergebnis der Berechnung zurückgeben.

Beispielausgabe: `2^2 = 4`

In [ ]:
```
# HIER IST PLATZ FÜR DEINE LÖSUNG! :)
```

Aufgabe 2 - Kreisfläche

Implementiere eine Funktion, die die Fläche eines Kreises berechnet.

Runde dein Ergebnis auf zwei Nachkommastellen.

Als Parameter soll der Radius in der Einheit Meter übergeben werden.

Mit Hilfe der `math`-Bibliothek und `math.pi` erhälst du im Programm einen möglichst exakten Wert für π.

Mit Hilfe von `round(Wert, Stellen)` kannst du einen Wert auf eine gewisse Anzahl an Nachkommastellen runden.

Die Funktion ist nur für die Berechnung zuständig. **Vor** dem Funktionsaufruf soll der Nutzer den Radius eingeben. Nach dem Funktionsaufruf soll die Ausgabe in der folgenden Form erfolgen:
`Die Fläche eines Kreises mit dem Kreisradius r = 5 m entspricht 78.54 m².`

In [ ]:
```
# HIER IST PLATZ FÜR DEINE LÖSUNG! :)
```

Aufgabe 3 - Kelvin und Fahrenheit

Zu implementieren sind 3 Funktionen.

Der Nutzer wird zunächst aufgefordert eine Temperatur in der Einheit $^\circ C$ einzugeben.

Im Anschluss gibt der Nutzer die gesuchte Einheit K oder $^\circ F$ ein.

Zu implementieren sind:

- Funktion 1: Nimmt die Temperatur und die gesuchte Einheit als Parameter entgegen und entscheidet, welche Berechnung ausgeführt werden muss.
- Funktion 2: Umrechnung von $^\circ C$ in K.
- Funktion 3: Umrechnung von $^\circ C$ in $^\circ F$.

Die Funktionen für die Umrechnung sollen den **Zahlenwert UND die Einheit** ausgeben - also eine Zeichenkette.

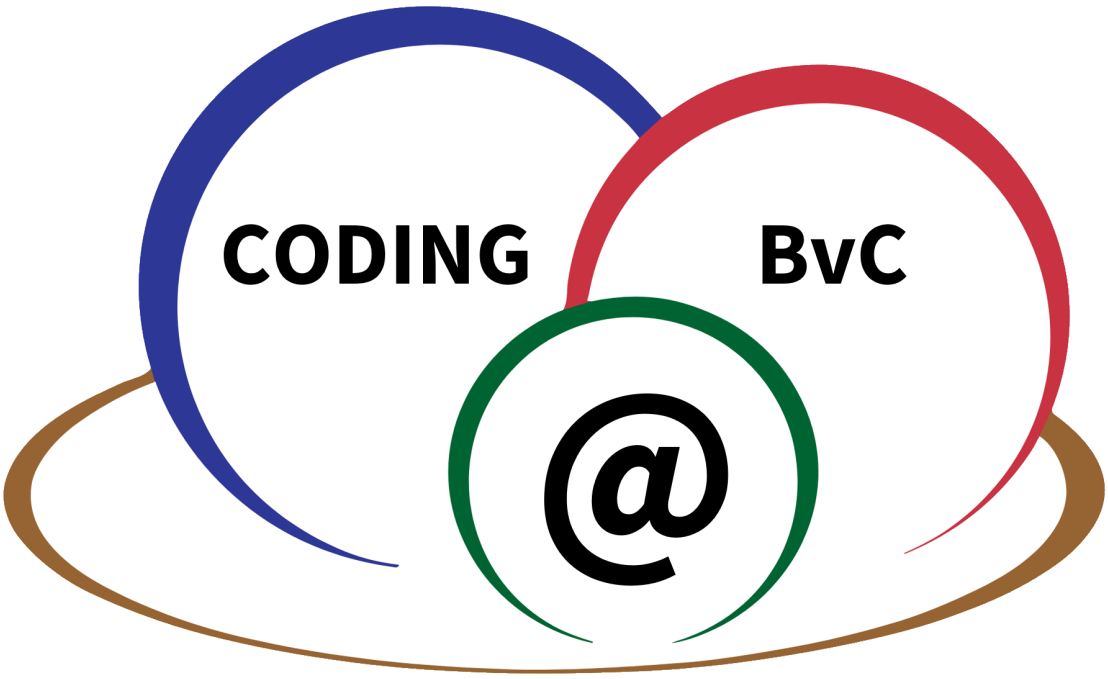
In [ ]:
```
# HIER IST PLATZ FÜR DEINE LÖSUNG! :)
```

Aufgabe 4 - Schere, Stein, Papier

Implementiere eine Funktion, die es ermöglicht, das ein Nutzer **"Schere, Stein, Papier"** gegen den Computer spielen kann.

- Die Wahl des Programms soll durch eine Zufallszahl geschehen: `1 == Stein`, `2 == Papier`, ...
- Es soll ausgegeben werden, was der Nutzer und das Programm gewählt haben.
- Implementiere alle Ausgänge: Unentschieden, Sieg und Niederlage
- Gewonnen hat man zum Beispiel, wenn der Nutzer Stein und das Programm Schwere wählt. Implementiere auch alle übrigen Fälle.

In [ ]:
```
# HIER IST PLATZ FÜR DEINE LÖSUNG! :)
```


CODING
BvC
@