Kapitel 4: Algorithmische Grundstrukturen und ihre Darstellung

Bisher haben wir hauptsächlich Programme gesehen, bei denen der Programmcode Zeile für Zeile nacheinander ausgewertet wird. Mit Hilfe **algorithmischer Grundstrukturen** können wir diesem Verhalten abweichen.

Algorithmische Grundstrukturen steuern dabei den Ablauf eines Programms. Mit ihnen könnt ihr Anweisungsblöcke definieren, die nur unter einer bestimmten Bedingung ausgeführt werden oder auch solche, deren Ausführung mehrfach erfolgt. Hierfür unterscheiden wir **Sequenzen**, **Verzweigungen** und **Schleifen**, welche wir uns - inklusiv dreier **Darstellungsformen** - nachfolgend genauer anschauen.

4.8. Schleifen / Iteration

Betrachten wir zunächst das nachfolgende Code-Beispiel:

In [ ]:
```
print(1)
print(2)
print(3)
print(4)
print(5)
print(6)
print(7)
print(8)
print(9)
print(10)
print(11)
print(12)
```

Was passiert hier? Zwölfmal wird immer und immer wieder die gleiche Funktion aufgerufen. Der Code ist nahezu identisch. Lediglich die ausgegebene Zahl ändert sich.

Jetzt stellen wir uns einmal vor, dass wir die Zahlen von 1 bis 100 ausgeben wollen. 100 mal den gleichen Code schreiben? Klingt nicht so spaßig? Stimmt! Dafür gibt es aber eine algorithmische Lösung: **die Schleife**!

Die Schleife als algorithmische Grundstruktur ermöglicht es uns, Code **mehrfach hintereinander auszuführen**, ohne immer und immer wieder die gleichen Code-Zeilen schreiben zu müssen. Das wiederholte Ausführen von Anweisungen wird bei der Programmierung als **Iteration** bezeichnet. Eine Iteration ist ein einzelner Durchlauf innerhalb einer Schleife. Wenn also eine Schleife viermal ausgeführt wird, dann besteht sie aus vier Iterationen.

Wir unterscheiden grundsätzlich **3 Arten von Schleifen**:

- Kopfgesteuerte Schleife
- Fußgesteuerte Schleife
- Zählschleife

Diese werden nachfolgend näher betrachtet.

4.8.1. Kopfgesteuerte Schleife

Bei der kopfgesteuerten Schleife wird eine Sequenz so lange wiederholt ausgeführt, bis eine Bedingung (im Kopf der Schleife) **nicht mehr erfüllt** ist.

Im **Pseudocode** könnte man exemplarisch folgendes formulieren:

```
SOLANGE zaehler < 5
    ergebnis = ergebnis + 1
    zaehler = zaehler + 1
```

Gut zu sehen ist, dass die letzten zwei Zeilen eingerückt sind. Das verdeutlicht, dass alles, was eingerückt ist, von der Schleife wiederholt wird.

Nachfolgend ist dieses Beispiel als **Code und Struktogramm** visualisiert. Zu sehen ist bei den Beispielen auch, dass vor der Schleife die Variablen `zaehler` und `ergebnis` jeweils mit dem Wert `0` initialisiert werden.

In **JEDEM** Schleifendurchlauf wird der `zaehler` um eins **inkrementiert** (das Fachwort für: 'erhöht'), solange bis die Bedingung im Schleifenkopf **nicht mehr erfüllt** ist.

Programmcode:

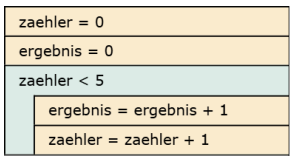
In [1]:
```
zaehler = 0
ergebnis = 0
while zaehler < 5:
    ergebnis = ergebnis + 1
    zaehler = zaehler + 1
```

Programmcode mit Ausgaben zum Verständnis:

In [2]:
```
zaehler = 0
ergebnis = 0
print(f'Vor der Schleife >>> zaehler = {zaehler} und ergebnis = {ergebnis}')
while zaehler < 5:
    print(f'    In der Schleife >>> zaehler = {zaehler} und ergebnis = {ergebnis}')
    ergebnis = ergebnis + 1
    zaehler = zaehler + 1
```

```
Vor der Schleife >>> zaehler = 0 und ergebnis = 0
    In der Schleife >>> zaehler = 0 und ergebnis = 0
    In der Schleife >>> zaehler = 1 und ergebnis = 1
    In der Schleife >>> zaehler = 2 und ergebnis = 2
    In der Schleife >>> zaehler = 3 und ergebnis = 3
    In der Schleife >>> zaehler = 4 und ergebnis = 4
```

Struktogramm:



In die geometrische Form für die Schleife (grüne Form) wird lediglich die Bedingung eingetragen. Dass es sich dabei um eine Schleife handelt, kann ja aus der Form abgeleitet werden.

Hinweise zum Code einer kopfgesteuerten Schleife:

- Jede kopfgesteuerte Schleife beginnt mit dem Schlüsselwort `while`.
- Danach folgt eine Bedingung, die wahr oder falsch sein kann. In PYTHON bedeutet dies, dass sie als Ergebnis den Wert `True` oder `False` haben muss. Nur im Falle von True werden die nun folgenden eingerückten Anweisungen auch abgearbeitet.
- Der Doppelpunkt `:` schließt die Bedingung ab.
- Alle Anweisungen, die bei Erfülltsein der Bedingung abgearbeitet werden sollen, müssen gleich weit eingerückt sein. Dies können auch mehrere Anweisungen sein. Nutze Tabulatoren (die Taste mit den Doppelpfeilen am linken Rand der Tastatur) zum Einrücken. Man bezeichnet diese Anweisungen als Schleifenkörper.
- Eine Schleife, d. h. der Schleifenkörper, darf niemals leer sein. Es muss mindestens eine eingerückte Anweisung vorhanden sein. Kommentarzeilen zählen dabei natürlich nicht.
- Die erste nicht eingerückte Anweisung kennzeichnet, dass die Schleife zu Ende ist.
- Damit die Schleife nicht zu einer Endlosschleife wird, muss sichergestellt werden, dass die Bedingung irgendwann im Laufe des Programmablaufs nicht mehr erfüllt ist.

Einzelne Schleifendurchläufe:

Schauen wir uns das Ganze zum besseren Verständnis noch etwas genauer an. Wir betrachten dazu das folgende Beispiel:

In [ ]:
```
i = 0
while i < 10:
    i = i + 1
```

Schritt	Bedingung i < 10?	Aktion	neuer Wert von i
0️⃣	-	Initialisierung (vor der eigentlichen Schleife)	i = 0
1️⃣	✅ WAHR (0 < 10)	i = 0 + 1 ➡️ 1	i = 1
2️⃣	✅ WAHR (1 < 10)	i = 1 + 1 ➡️ 2	i = 2
3️⃣	✅ WAHR (2 < 10)	i = 2 + 1 ➡️ 3	i = 3
...	...	...	...
9️⃣	✅ WAHR (8 < 10)	i = 8 + 1 ➡️ 9	i = 9
🔟	✅ WAHR (9 < 10)	i = 9 + 1 ➡️ 10	i = 10
⏹️	❌ FALSCH (10 < 10)	Schleife wird verlassen	i = 10 (Endwert)

Die Bedingung `i < 10` wird vor jedem Durchlauf geprüft. Bei `i = 10` ist `10 < 10` falsch und die Schleife bricht ab.

4.8.2. Übungen zur kopfgesteuerten Schleife

Aufgabe 1

Implementiere den folgenden Pseudocode.

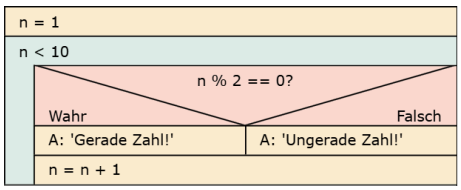
```
Eingabe einer Zahl n
Setze die Variable Zähler auf 1
SOLANGE der Zähler kleiner n ist
    Erhöhe Zähler um 0,5
Ausgabe des Zählers
```

In [2]:
```
n = input('Gib eine Zahl n ein: ')
n = float(n)

zaehler = 1

while zaehler < n:
    zaehler = zaehler + 0.5

print(f'Wert der Zähler-Variable bei Eingabe von n = {n}: {zaehler}')
```

```
Wert der Zähler-Variable bei Eingabe von n = 5.0: 5.0
```

Aufgabe 2

Implementiere das folgende Struktogramm.



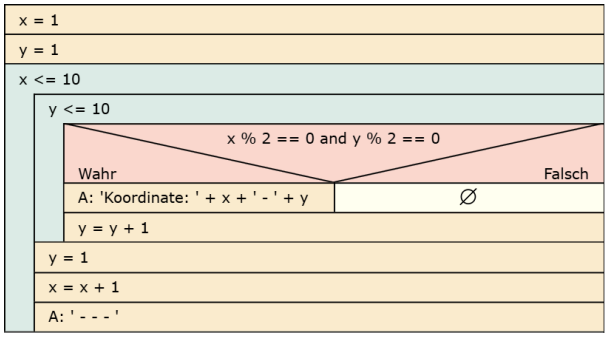
Besonders zu beachten ist, dass die letzte Anweisung `n = n + 1` über die gesamte Breite geht. Das bedeutet, dass diese Anweisung **NICHT MEHR** zur Verzweigung gehört!

In [3]:
```
n = 1

while n < 10:
    if n % 2 == 0:
        print('Gerade Zahl!')
    else:
        print('Ungerade Zahl!')
    n = n + 1
```

```
Ungerade Zahl!
Gerade Zahl!
Ungerade Zahl!
Gerade Zahl!
Ungerade Zahl!
Gerade Zahl!
Ungerade Zahl!
Gerade Zahl!
Ungerade Zahl!
```

Aufgabe 3

Implementiere das folgende Struktogramm.



Hinweis: Damit die Ausgabe innerhalb der Verzweigung fehlerfrei funktioniert muss der Datentyp der Variablen `x` und `y` entsprechend konvertiert werden.

In [ ]:
```
x = 1
y = 1
while x <= 10:
    while y <= 10:
        if x % 2 == 0 and y % 2 == 0:
            print(f'Koordinate: {x} - {y}')
        y = y + 1
    y = 1
    x = x + 1
    print('- - -')
```

```
- - -
Koordinate: 2 - 2
Koordinate: 2 - 4
Koordinate: 2 - 6
Koordinate: 2 - 8
Koordinate: 2 - 10
- - -
- - -
Koordinate: 4 - 2
Koordinate: 4 - 4
Koordinate: 4 - 6
Koordinate: 4 - 8
Koordinate: 4 - 10
- - -
- - -
Koordinate: 6 - 2
Koordinate: 6 - 4
Koordinate: 6 - 6
Koordinate: 6 - 8
Koordinate: 6 - 10
- - -
- - -
Koordinate: 8 - 2
Koordinate: 8 - 4
Koordinate: 8 - 6
Koordinate: 8 - 8
Koordinate: 8 - 10
- - -
- - -
Koordinate: 10 - 2
Koordinate: 10 - 4
Koordinate: 10 - 6
Koordinate: 10 - 8
Koordinate: 10 - 10
- - -
```

Aufgabe 4

Implementiere ein Programm, welches nach Eingabe einer oberen Grenze, alle Zahlen von 0 bis n aufaddiert.

Bei der Eingabe der Zahl 3 ergibt sich beispielsweise das Ergebnis 6.

Gerechnet wird dabei: $0 + 1 + 2 + 3 = 6$.

In [ ]:
```python
obere_grenze = input('Gib eine obere Grenze ein: ')
n = int(obere_grenze)

zaehler = 0
summe = 0

while zaehler <= n:
    summe = summe + zaehler
    zaehler = zaehler + 1
```

```
print(f'Die Quersumme von 0 bis {n} ergibt sich zu: {summe}')
```

```
Die Summe von 0 bis 3 ergibt sich zu: 6
```