Schulung Blinkenlights

Komplett ohne irgendwas Blinkendes

Sebastian Köll, Jan Rüther

August 17, 2020

1 Aufgabe 1: Python zum Fliegen bringen

1.1 Aufgabe 1a

Bringt das folgende Programm zum laufen und testet es

```
import math


def main():

    radius = int(input("Kugelradius in Metern angeben;"))

    oberflaeche = 4 * math.pi * radius**2
    volumen = (4/3) * math.pi * radius**3

    print(f" Die Obefleache der Kugel betraegt = {oberflaeche}
                                        Quadratmeter")
    print(f" Das Volumen der Kugel betraegt = {volumen} Kubikmeter"
                                        )


if __name__ == '__main__':
    main()
```

1.2 Aufgabe 1b

Schreibt ein Programm (ähnlich dem in Aufgabe 1a), welches die Höhe h und den Radius r eines Zylinders einliest und dann die gesamte Oberfläche und das Volumen des Zylinders berechnet.

1.3 Aufgabe 1c

Schreibt ein Programm, welches für eine positive ganze Zahl die Quersumme berechnet. Die Quersumme von 1337 bespielsweise ist 14.

2 Aufgabe 2: Dinge mit Listen und Schleifen

Schreibt ein Programm, das eine Folge von Komma-Zahlen einliest. Das Ende der Zahlenfolge wird erkannt durch das erste Zeichen, welches keine Zahl (also z.B. ein Buchstabe)ist. Das Programm soll dann für die eingelesenen Zahlen folgende Größen berechnen:

a. Die Anzahl der eingelesenen Zahlen,

b. Die Summe der eingelesenen Zahlen,

c. Das Maximum der eingelesenen Zahlen,

d. Das Minimum der eingelesenen Zahlen,

e. Den Mittelwert der eingelesenen Zahlen

3 Struktogramme

Beim Programmieren ist es häufig hilfreich, sich vorab die Zeit zu nehmen und das Programm durchzuplanen, bevor mit der eigentlichen Programmierung angefangen wird. Um Programme zu planen existiert eine Vielzahl an grafischen Modellierungshilfen. Diese sind dazu da, Programmaufbau und Ablauf in grafischer Form übersichtlich darzustellen. Eine Möglichkeit hierfür sind so genannte Struktogramme (auch Nassi-Schneiderman Diagramme).
Das Diagramm besteht aus verschiedenen Elementen, welche die Elemente eines Programms widerspiegeln. Für uns sind zunächst die von Python unterstützen Elemente interessant.

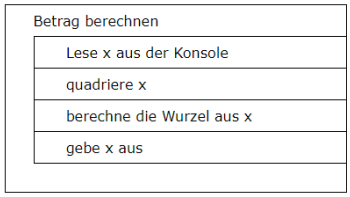
Figure 1: Weeks of coding...

3.1 Symbolik

im Folgenden wird de Symbolik von Struktogrammen kurz anhand von Minimalbeispielen erklärt. Diese Beispiele enthalten immer ein kleines Struktogramm mit dem passenden Codesnippet.

3.1.1 Anweisung

Eine Anweisung wird in einem rechteckigen Kasten geschrieben. Alle die Blöcke werden von oben nach unten abgearbeitet. Im folgenden Beispiel wird der Betrag einer Zahl berechnet und ausgegeben, welche zuvor eingelesen wurde.

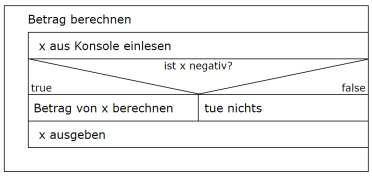



```
from math import sqrt

x = int(input("x eingeben: "))
x = x**2
x = sqrt(x)
print(x)
```

3.1.2 If-Abfrage

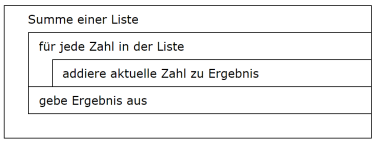
Eine If-Abfrage[1] stellt eine Verzweigung im Programm dar. Wenn eine Bedingung erfüllt ist, wird en bestimmter Teil Code ausgeführt.
Wir lesen eine Zahl ein. Falls die Zahl negativ ist, soll der Betrag berechnet werden. Abschließend wird die Zahl ausgegeben:




```
x = int(input("x eingeben: "))
if x < 0:
  x = abs(x)
print(x)
```

3.1.3 For-Schleife

For-Schleifen werden da verwendet, wo vorab bereits bekannt ist, wie viele Schleifendurchläufe gebraucht werden. Beispiele hierfür ist das iterieren über eine Liste oder das durchführen eines Vorgangs n-mal.
Wir nehmen an, dass in einer Liste eine Menge an Zahlen steht. Die For-Schleife soll die Zahlen aufsummieren. Abschließend soll die Summe ausgegeben werden:




```
num_lst = [1, 2, 3, 4, 5]
res = 0
for num in num_lst:
  res = res + num
print(res)
```

3.1.4 While-Schleife

While-Schleifen werden dort verwendet, wo die Abbruchbedingung der Schleife anfangs noch nicht direkt bekannt ist. Die Schleife wird so lange ausgeführt, wie die Bedingung im Schleifenkopf True ist.

[1] Achtung: Es wird IF-Abfrage genannt. If-Schleifen gibt es nicht!!!

3.2 Aufgaben

Die folgenden Aufgaben könnt ihr mit Hilfe des Tools *structorizer* (https://www.structorizer.com/struct.php) bearbeiten. Das Tool weist weitere Elemente für Struktogramme auf, die wir entweder nicht verwenden oder die nicht in Python vorhanden sind.

3.2.1 Warmup

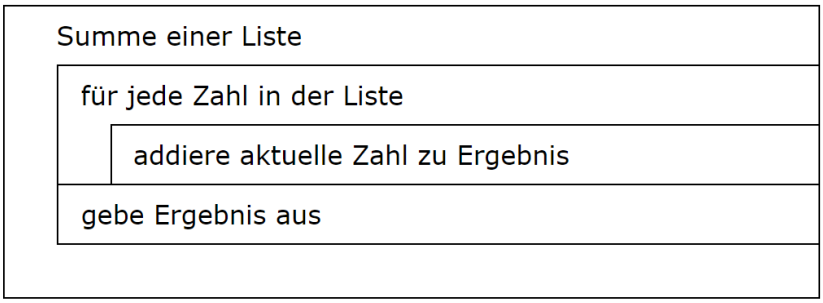
Um warm zu werden mit dem Tool, sollt ihr zunächst die Struktogramme aus 3.1 nachbauen.

3.2.2 Code zu Struktogramm

Erstellt aus eurem Code für die Aufgabe 2 (Dinge mit Listen und Schleifen) ein Struktogramm.

3.2.3 Struktogramm zu Code

Erstellt zu folgendem Struktogramm das Programm:



3.2.4 Würfeln

Es soll ein Programm entworfen werden, welches einen Würfel so lange würfelt, bis eine Zahl 3 mal hintereinander gefallen ist. Die Anzahl der benötigten Würfe ist am Ende auszugeben. Der Würfel hat 6 Seiten.
Erstellt zunächst das Struktogramm und implementiert eure Lösung anschließend.

3.2.5 Zahlenspiel

Ihr sollt ein kleines Zahlenspiel planen und implementieren. Das Spiel wird über die Konsole gespielt. Der Spieler muss eine Zahl (ganze Zahl) erraten, die der Computer sich zuvor ausdenkt. Hierbei wird die vom Spieler geratene Zahl über die Konsole eingelesen. Der Computer gibt daraufhin den Tipp, ob die geratene Zahl kleiner oder größer ist als die zu erratende Zahl. Sollte der Spieler die Zahl richtig erraten, wird das Programm mit einer passenden Ausgabe beendet.

Erstellt zunächst das Struktogramm des Spiels. Anhand dieses Struktogramms sollt ihr anschließend das Programm implementieren.

4 Aufgabe 3: Streichhölzer ziehen

In dieser Aufgabe soll ein bekanntes Streichholzspiel realisiert werden. Von einer Anfangsmenge von Streichhölzern nehmen zwei Spieler abwechselnd eins, zwei oder drei Hölzchen weg. Wer das letzte Hölzchen nehmen muß, hat verloren. Das Programm soll so gestaltet sein, daß ein Spieler gegen den Computer spielt.

Der Spieler kann dabei bei jedem seiner Zügen nach Gutdünken wahlweise eins, zwei oder drei Hölzchen nehmen. Die Züge des Computers seien zunächst in einer ersten Version so realisiert, daß er zufällig eins, zwei oder drei Hölzchen nimmt. Lediglich wenn nur noch vier Hölzchen oder weniger übrig geblieben sind, spielt der Computer so, daß er gewinnt. Bei vier Hölzchen beispielsweise nimmt er drei, damit der Spieler auf dem letzten Hölzchen sitzen bleibt.

Zufallszahlen kann man in python mit der Funktion **randint** aus der Bibliothek **random** generieren. Mit den Anweisungen:

```
import random

krasse_zahl = random.randint(1,10)
```

wird der Variable krasse_zahl eine Zufallszahl zwischen 1 und 10 zugewiesen.

Damit bei jedem Programmdurchlauf immer eine neue Folge von Zufallszahlen berechnet wird, muss der Zufallsgenerator einmal am Anfang des Programms initialisiert werden. Dies geschieht mit der Funktion **random.seed** Mit könnte man somit beispielsweise Zufallszahlen zwischen 1 und 3 erzeugen. Achte darauf, daß auch der Spieler immer nur eins, zwei oder drei Hölzchen nehmen darf, nicht mehr und nicht weniger.

Wenn das Programm läuft, überlegt euch eine optimale Strategie für den Computer. Der Computer soll also in einer verbesserten Version des Programms von Beginn an stehts den optimalen Zug ausführen. Er soll also immer so ziehen, daß er, wenn er die Chance zu gewinnen hat, diese auch nutzt. Die Darstellung auf der Console könnte dann z.B. wie folgt aussehen:

```
Die aktuelle Anzahl der Hoelzchen ist: 13
* * * * * * * * * * * * *
I I I I I I I I I I I I I
I I I I I I I I I I I I I
I I I I I I I I I I I I I
* * * * * * * * * * * * *
Gönn dir Hoelzchen, Brudi! (1, 2, 3):
```

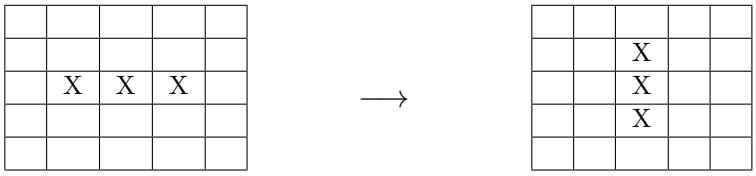
5 Conways *game of life*

Im Jahr 1968 wurde von J. H. Conway an der Universität Cambridge das *game of life* erfunden und 1970 von M. Gardner im Scientific American einem breiten Publikum vorgestellt. Dabei handelt es sich um einen Algorithmus, der das Wachstum von fiktiven Lebewesen (Bakterien) simuliert. Infolge der interessanten Muster, die dabei entstehen, ist das game of life weit über Biologenkreise hinaus bekannt geworden. Es ist ein Beispiel für einen sogenannten zellulären Automaten. Schauplatz des *game of life* ist ein zweidimensionales Gitter aus Zellen, die entweder tot (' ') oder lebendig ('X') sind. Wie sich eine Zelle weiter entwickelt, hängt von ihren acht Nachbarn ab (diagonale Nachbarn zählen ebenfalls), und zwar gelten folgende Regeln:

1 Eine lebende Zelle überlebt in der nächsten Generation, wenn sie zwei oder drei Nachbarn hat. Sind es weniger bzw. mehr, so stirbt sie an Vereinsamung bzw. Überbevölkerung.

2 Eine tote Zelle wird immer dann in der nächsten Generation zum Leben erweckt, wenn sie genau drei lebendige Nachbarn hat, ansonsten bleibt sie tot.

Die Zeit verstreicht dabei in diskreten Schritten, d.h. jede Zelle verharrt in ihrem zuvor eingenommenen Zustand, bis gewissermaßen bei einem Gongschlag alle gleichzeitig in den neuen Zustand übergehen. Anders ausgedrückt: Es wird für jede Zelle nachgeschaut, wie ihr Zustand und der ihrer Nachbarn zu einer bestimmten Zeit n ist und berechnet, wie ihr Zustand zur nächsten Zeit $n + 1$ sein wird. Hat man dies für alle Zellen getan, so werden alle gleichzeitig auf den neuen Zustand gesetzt.
Zur Veranschaulichung ist dies im Folgenden an einem einfachen Beispiel erläutert. Das *Spielfeld* besteht hierbei aus einem 5 mal 5 Zellen großen Feld.



Generation 0 Generation 1

Die 0-te Generation besteht aus einer Linie von drei belebten Feldern. Für die Folgegeneration geht unter Annahme der Regel 1 hervor, dass die beiden äußeren Zellen sterben werden, da diese weniger als 2 Nachbarn haben. Die mittlere Zelle überlebt, da sie 2 Nachbarn hat und somit nicht mehr als 3 oder weniger als 2.
Wird die Regel 2 angewandt, so entstehen zwei neu belebte Zellen über und unter der mittleren Zelle, da diese jeweils genau drei Nachbarn haben. Die restlichen Zellen bleiben unberührt.

Verschiedene Startmuster können hierbei zu unterschiedlichen "Populationsverläufen" führen.

5.1 Aufgabenbeschreibung

Es soll das *game of life* in Python selber Programmiert werden. Eine erste grobe Aufgabenaufteilung kann hierbei aus dem *EVA*-Prinzip abgeleitet werden:

Eingabe Als Eingabe wird hierbei das initiale Füllen des Spielfelds verstanden. Die 0-te Generation soll von Außen vorgegeben werden können. Die eingegebenen Daten müssen in einer Form vorliegen, dass die Verarbeitung sie für die Ermittlung der Nachfolgegenerationen verwenden kann.

Verarbeitung Die Verarbeitung ist der Motor des Spiel. Hierbei müssen die oben genannten Regeln umgesetzt werden und Generation für Generation ermittelt werden. Das Weiterschalten von den einzelnen Generationen kann über eine Zeit oder einen Tastendruck geschehen. Die ermittelten Daten sollen am Ende einer Generationsermittlung immer in der Form vorliegen, wie sie auch von der Eingabe zur Verfügung gestellt wurden. In dieser Form werden sie der Ausgabe zur Verfügung gestellt.

Ausgabe Unter der Ausgabe ist die Darstellung des aktuellen Spielfelds samt "Bevölkerung" zu verstehen.

Vorab sollten sich jedoch einige Gedanken über die Rahmenbedingungen wie Größe des Spielfelds und Regeln für das Verhalten der Zellen am Rand des Spielfelds gemacht werden.
Mehr Informationen zum *game of life* gibt es mit Sicherheit im Internet! Im Folgenden werden wir uns mit der groben Vorgehensweise zum Implementieren des Programms beschäftigen.

5.2 Vorschläge für die Umsetzung

5.2.1 Datenformat einer Generation

Eine der wichtigsten Dinge die bei der Konzeption eines Programms beachtet werden müssen sind die Schnittstellen der einzelnen Module untereinander. Dies wird umso wichtiger, wenn mehrere Programmiere parallel an einem Projekt an unterschiedlichen Modulen arbeiten. Als Module können hierbei die einzelnen Schritte des EVA-Prinzips angesehen werden. Die Schnittstelle is in diesem Fall das weiterreichen der Daten. Wird hier nicht von Anfang an eine einheitliche Form gewählt, so führt dies beim Zusammenführen der Module zu einem großen Chaos und eine Menge zusätzliche Arbeit entsteht.
Was als Daten übergeben werden muss, ist im Grunde genommen nur das Spielfeld. Dieses besteht aus einzelnen Zellen, welche die Werte *tot* oder *lebendig* annehmen können. Da das Spielfeld eine Fläche ist, bietet sich eine zweidimensionale Datenstruktur an. Eine Umsetzung hierfür kann in Python ein Numpy Array sein:

```
import numpy as np
height = 4
width = 5
my_field = np.ones((height, width))
```

Das Codesnippet erzeugt ein Array der gewünschten Höhe und Breite, welches initial mit Nullen gefüllt ist:

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Um nun die zwei verschiedenen Zustände (*lebendig*, *tot*) anzuzeigen, sind verschiedene Darstellungen denkbar. Zum einen könnten tote Zellen mit Nullen und lebendige Zellen mit Einsen dargestellt werden. Denkbar wäre auch eine boolsche Darstellung True für lebendig und Fals für tot oder eine Darstellung mit Hilfe von Strings:

0	1	0	1	0
0	0	1	0	0
0	1	0	1	0
0	0	0	0	0

Darstellung durch Nullen und Einsen

False	True	False	True	False
False	False	True	False	False
False	True	False	True	False
False	False	False	False	False

Darstellung durch True oder False

tot	leb	tot	leb	tot
tot	tot	leb	tot	tot
tot	leb	tot	leb	tot
tot	tot	tot	tot	tot

Darstellung durch Strings

Vorab sollte in der Gruppe festgelegt werden, welches Datenformat ihr wählt. Die oben aufgeführten Formate sind nur Vorschläge. Falls ihr bessere Ideen habt, könnt ihr diese auch gerne umsetzen.

5.2.2 Eingabe der Generation 0

Für die Eingabe der nullten Generation sind verschiedene Möglichkeiten Denkbar. Ziel des Ganzen ist es, am ende der Eingabe ein Spielfeld in der Form vorliegen zu haben, wie es in 5.2.1 festgelegt wurde. Hier sollte die Möglichkeit bestehen, einzelne Zellen als *lebendig* zu initialisieren. Denkbar wäre auch das direkte Setzen verschiedener Formen.
Als simpelste Form der Eingabe ist hierbei sicher das hardgecodete initiale setzen des Spielfelds möglich. Denkbar wäre auch das einlesen einer *csv* Datei, in der das Spielfeld eingetragen ist. Ein Beispiel für eine solche Datei ist in dem Ordner **TODO** zu finden.
Etwas fortgeschrittenere Programmierer können hierbei auch auf eine grafische Oberfläche zurückgreifen, eine so genannte *GUI*.

5.2.3 Verarbeitung der einzelnen Generationen

5.2.4 Ausgabe / Anzeige einer Generation

Bei der Anzeige der aktuellen Generation kann im simpelsten Fall die Konsole von Python verwendet werden. Denkbar wäre hier die Darstellung in Textform:
6
7
8
9
0
1 X X
2 X X
3 X XX X
4 XX X X X XXXX X
5 XX X XX XXXX X
6 X X X X XX
7 X X XX X XX
8 X X X X
9 X
0
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789