Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Marius Weidner, Hannes Saffrich
Lukas Kleinert, Timpe Hörig

Universität Freiburg
Institut für Informatik
Wintersemester 2023

Übungsblatt 11

Abgabe: Montag, 15.01.2024, 9:00 Uhr morgens

Aufgabe 11.1 (Generatoren; 10 Punkte; Datei: `generators.py`)

In dieser Aufgabe dürfen Sie keine Generatoren zu Listen umwandeln, da dies gerade den Vorteil von Generatoren zunichte macht.

(a) `collatz`; **2.5 Punkte**

Schreiben Sie eine Funktion `collatz`, die eine ganze, positive[1] Zahl `n` als Argument nimmt und einen Generator zurückgibt, der nacheinander die Ganzzahlen c_i der rekursiven Collatz-Folge für $i \in \mathbb{N}_0$ beginnend mit $c_0 = \texttt{n}$ produziert:

$$c_{i+1} = \begin{cases} \frac{c_i}{2}, & \text{wenn } c_i \text{ gerade} \\ 3 \cdot c_i + 1, & \text{wenn } c_i \text{ ungerade} \end{cases}$$

Ist $c_i = 1$, soll der Generator terminieren.

```
assert list(collatz(11)) == [
    11, 34, 17, 52, 26, 13, 40,
    20, 10, 5, 16, 8, 4, 2, 1
]
assert list(collatz(0)) == list(collatz(-4)) == []
assert len(list(collatz(97))) == 119
```

(b) `random`; **2.5 Punkte**

Schreiben Sie eine Funktion `random`, die einen Generator zurückgibt, der pseudozufällige[2] Zahlen generiert. Die Funktion soll dazu vier ganzzahlige Argumente haben: `seed`, `a`, `b` und `m`, und einen Generator zurückgeben, der nacheinander die Ganzzahlen y_i der (unendlichen) Folge

$$y_{i+1} = (a \cdot y_i + b) \; mod \; m \quad \text{für } i \in \mathbb{N}_0 \text{ und } y_0 = \texttt{seed}$$

produziert. Wenn Sie unterschiedliche Werte für die Attribute `seed`, `a`, `b` und `m` ausprobieren, werden Sie feststellen, dass einige Werte bessere Ergebnisse liefern als andere. Welche Werte besonders sinnvoll sind, können Sie bei Interesse zum Beispiel hier herausfinden: https://de.wikipedia.org/wiki/Kongruenzgenerator#Linearer_Kongruenzgenerator.

```
assert list(zip(range(17), random(11, 5, 0, 64))) == list(zip(range(17), [
    11, 55, 19, 31, 27, 7, 35, 47, 43, 23, 51, 63, 59, 39, 3, 15, 11
]))
```

[1] Ist die Zahl nicht positiv, soll nichts generiert werden.
[2] https://de.wikipedia.org/wiki/Pseudozufall

(c) `chunks`; **2.5 Punkte**
Schreiben Sie eine Funktion `chunks`, die einen Iterator `iter` und eine natürliche Zahl `n` als Argumente nimmt und einen Generator zurückgibt. Dieser Generator produziert Listen, die jeweils die nächsten `n` Element aus `iter` beinhalten. Gibt es weniger als `n` Elemente, soll eine kürzere Liste zurückgeben werden. Wenn `n` gleich 0 ist, werden unendlich viele leere Listen produziert.

```
assert list(chunks(iter(range(1, 9)), 3)) == [
    [1, 2, 3], [4, 5, 6], [7, 8]
]
assert list(chunks(iter(range(1, 5)), 2)) == [[1, 2], [3, 4]]
assert list(chunks(iter([]), 1)) == []
assert next(chunks(iter([]), 0)) == next(chunks(iter([1, 2]), 0)) == []
```

(d) `flatten`; **2.5 Punkte**
Schreiben Sie eine Funktion `flatten`, die einen Iterator von Listen `iter` als Argument nimmt und einen Generator zurückgibt, der zuerst die Elemente aus der ersten Liste produziert, anschließend die Elemente aus der zweiten Liste und so weiter...

```
assert list(flatten(iter([[1, 2, 3], [4, 5], [6]]))) == list(range(1, 7))
assert list(flatten(iter([[]]))) == []
```

Aufgabe 11.2 (Graphen; 10 Punkte, Datei: `graphs.py`)

In dieser Aufgabe betrachten wir Dictionaries der Form `dict[T, set[T]]`. Ein solches Dictionary nennen wir genau dann einen 'Graph'[3], wenn jeder Wert in den Werte-Mengen im Dictionary auch ein Schlüssel im selben Dictionary ist.

(a) `is_graph`; **2.5 Punkte**
Schreiben Sie eine Funktion `is_graph`, die ein Dictionary `d` der Form `dict[T, set[T]]` als Argument nimmt und genau dann `True` zurückgibt, wenn `d` ein Graph ist.

```
example = {0: {1, 2}, 1: {2, 3}, 2: {0, 1, 2}, 4: {0}}
assert not is_graph(example)
example_graph = example | {3: set()}
assert is_graph(example_graph)
assert not is_graph({"a": {"a", "aa"}})
assert is_graph({})
```

(b) `to_graph`; **2.5 Punkte**
Schreiben Sie eine Funktion `to_graph`, die ein Dictionary `d` der Form `dict[T, set[T]]` als Argument nimmt und eine Kopie von `d` zurückgibt, die jedoch zu einem Graph ergänzt wurde. Fügen Sie dazu jeden Wert einer Werte-Menge von `d`, der kein Schlüssel von `d` ist, als Schlüssel mit leerer Werte-Menge in das Resultat ein.

[3] Graphen sind wichtige Datenstrukturen in der Informatik. Die Definition eines Graphen ist üblicherweise jedoch allgemeiner als die in dieser Aufgabe. Ein 'Graph' in dieser Aufgabe entspricht eher der Implementierung eines 'gerichteten Graphs ohne Mehrfachkanten'. Mehr dazu hier: https://de.wikipedia.org/wiki/Graph_(Graphentheorie)

```
assert to_graph(example) == to_graph(example_graph) == example_graph
assert to_graph(example_graph) is not example_graph
assert to_graph({"a": {"a", "aa"}}) == {"a": {"a", "aa"}, "aa": set()}
assert to_graph({}) == {}
```

(c) `nodes`, `edges`; **2.5 Punkte**
Die Schlüssel in einem Graphen nennen wir 'Knoten'. Jedes Tupel von Knoten `(a,b)`, bei dem `b` ein Element der Werte-Menge von `a` ist, bezeichnen wir als 'Kante'.

Schreiben Sie eine Funktion `nodes`, die einen Graph `graph` als Argument nimmt und einen Generator zurückgibt, der alle Knoten von `graph` produziert. Schreiben Sie auch eine Funktion `edges`, die ebenso einen Graph `graph` als Argument nimmt und einen Generator zurückgibt, der alle Kanten von `graph` produziert.

```
assert set(nodes(example_graph)) == {0, 1, 2, 3, 4}
assert len(list(nodes(example_graph))) == 5
assert set(nodes({})) == set()
assert set(edges(example_graph)) == {
    (0, 1), (0, 2), (1, 2), (1, 3),
    (2, 0), (2, 1), (2, 2), (4, 0)
}
assert len(list(edges(example_graph))) == 8
assert set(edges({})) == set()
```

(d) `invert_graph`; **2.5 Punkte**
Schreiben Sie eine Funktion `invert_graph`, die einen Graph `graph` als Argument nimmt und einen Graph vom gleichen Typ zurückgibt. Für jede Kante `(a,b)` in `graph` soll der invertierte Graph die Kante `(b,a)` besitzen. Ansonsten sollen keine weiteren Kanten (oder Knoten) vorkommen. Achten Sie jedoch insbesondere darauf, dass der invertierte Graph auch wirklich ein Graph ist!

```
assert invert_graph(example_graph) == {
    0: {2, 4}, 1: {0, 2}, 2: {0, 1, 2}, 3: {1}, 4: set()
}
assert invert_graph(invert_graph(example_graph)) == example_graph
assert invert_graph({"a": {"a"}}) == {"a": {"a"}}
assert invert_graph({}) == {}
```

(e) `has_cycle`; **0 Punkte** (Knobelaufgabe, **schwer**)
Einen Zyklus im Graph `graph` definieren wir als Folge von Knoten $o_1, o_2, \ldots, o_n$ aus `graph`, wobei Tupel aufeinanderfolgender Knoten eine Kante sind (also $\forall i \in \{1, \ldots, n-1\} : o_{i+1}$ `in graph`$[o_i]$), und (o_1 `==` o_n) gilt. Der Graph `example_graph` besitzt z.B. die Zyklen `(0,1,2,0)`, `(1,2,1)`, `(2,2)`. Die Folge `(0,2,1,0)` ist hingegen kein Zyklus in `example_graph`. Schreiben Sie eine Funktion `has_cycle`, die einen beliebigen Graph `graph` als Argument nimmt und zurückgibt, ob der Graph einen Zyklus besitzt.
Hinweis: Sie können eine rekursive Hilfsfunktion schreiben, die von einem gegebenen Knoten ausgehend Kanten folgt, und genau dann True zurückgibt, wenn ein bereits besuchter Knoten erneut besucht wird.

```
assert has_cycle(example_graph)
example_graph2 = {
    0: {1}, 1: {2}, 2: set(), 3: {0},
    4: {1, 5}, 5: {6}, 6: {7}, 7: {3, 8},
    8: set(), 9: {8}
}
assert not has_cycle(example_graph2)
assert has_cycle(example_graph2 | {3: {0, 4}})
assert has_cycle({"a": {"aa", "a"}, "aa": set()})
assert not has_cycle({1: {3}, 2: {1, 4}, 3: {4}, 4: set()})
assert not has_cycle({})
```

Aufgabe 11.3 (Erfahrungen; 0 Punkte; Datei: `NOTES.md`)

Notieren Sie Ihre Erfahrungen mit diesem Übungsblatt (benötigter Zeitaufwand, Probleme, Bezug zur Vorlesung, Interessantes, etc.).

Editieren Sie hierzu die Datei `NOTES.md` im Abgabeordner dieses Übungsblattes auf unserer Webplatform. Halten Sie sich an das dort vorgegebene Format, da wir den Zeitbedarf mit einem Python-Skript automatisch statistisch auswerten. Die Zeitangabe `6.5 h` steht dabei für 6 Stunden 30 Minuten.