Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Marius Weidner
Simon Dorer, Timpe Hörig

Universität Freiburg
Institut für Informatik
Wintersemester 2025

Übungsblatt 12

Abgabe: Montag, 19.01.2025, 9:00 Uhr

Aufgabe 12.1 (Generatoren; 10 Punkte; Datei: `generators.py`)

In dieser Aufgabe sollen Sie Generatoren definieren. Dabei dürfen Sie keine Generatoren zu Listen umwandeln, da dies gerade den Vorteil von Generatoren zunichte macht. Beachten Sie zusätzlich, dass Generatoren kein Indexing, Slicing, Längenabfragen, usw. unterstützen! Verwenden Sie für Generatoren die Typannotation `Iterator` aus dem Modul `typing`. Denken Sie daran die Typen der generierten Elemente anzugeben.

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

Die Collatz-Folge ist eine Folge natürlicher Zahlen, die wie folgt definiert ist:

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

Schreiben Sie eine Funktion `collatz`, die eine ganze Zahl `n` als Argument entgegennimmt und einen Generator zurückgibt, der die Elemente der Collatz-Folge beginnend bei $c_0 =$ `n` erzeugt. Der Generator soll die Erzeugung weiterer Werte beenden, sobald der Wert `1` erreicht wurde. Wird die Funktion mit einem Wert `n < 1` aufgerufen, so soll ein `ValueError` mit einer aussagekräftigen Fehlermeldung ausgelöst werden.

Zum Beispiel:

```
>>> list(collatz(10))
[10, 5, 16, 8, 4, 2, 1]
>>> list(collatz(1))
[1]
```

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

Schreiben Sie eine Funktion `random`, die einen Generator zurückgibt, welcher Pseudozufallsbitsequenzen nach der Linear-Feedback-Shift-Register-Methode erzeugt.[1] Die Funktion erhält einen String `seed` als Argument, der ausschließlich aus den Zeichen `"0"` und `"1"` besteht. Der String `seed` repräsentiert den Anfangszustand y_0 des Generators.

Aus der i-ten pseudozufälligen Bitsequenz y_i wird die nächste Bitsequenz y_{i+1} wie folgt erzeugt: Es werden das erste Bit, das mittlere Bit (bei gerader Länge abgerundet) und das letzte Bit von y_i ausgewählt. Diese drei Bits werden mittels `XOR` verknüpft. Das resultierende Bit wird rechts an die Bitfolge angehängt,

[1] https://de.wikipedia.org/wiki/Linear_rückgekoppeltes_Schieberegister

während alle übrigen Bits um eine Position nach links verschoben werden und das vorderste Bit entfernt wird. Die Gesamtlänge der Bitfolge bleibt dabei unverändert.

Hinweis: Die `XOR`-Operation ist wie folgt definiert:

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

Ein Beispiel:
Angenommen, der aktuelle Zustand des Generators ist $y_i =$ `1010111010`. Dann werden die Bits an den Positionen 0, 4 und 9 ausgewählt (also `1`, `1` und `0`). Da `1 XOR 1 XOR 0 = 0` gilt hängen wir das Bit `0` rechts an die Bitfolge an und verschieben alle übrigen Bits um eine Position nach links. Somit ergibt sich für den nächsten Zustand des Generators $y_{i+1} =$ `0101110100`.

Der Aufruf der Funktion mit dem Seed `"1010111010"` soll also die folgende Bitfolge generieren:

```
>>> r = random("1010111010")
>>> for _ in range(10): print(next(r), end=" ")
...
0101110100 1011101001 0111010011 1110100111 1101001111 1010011110
↪  0100111101 1001111010 0011110100 0111101001
```

(c) `stop_if_double`; **2.5 Punkte**
Schreiben Sie eine Funktion `stop_if_double`, die einen Iterator `it` als Argument entgegennimmt und einen Generator zurückgibt, der solange Elemente aus `it` erzeugt, bis ein Wert unmittelbar zweimal hintereinander auftritt. In diesem Fall soll der entsprechende Wert nur einmal erzeugt werden.

```
>>> s1 = stop_if_double(iter(range(5)))
>>> list(s1)
[0, 1, 2, 3, 4]
>>> s2 = stop_if_double(iter("Hallo Welt :)"))
>>> list(s2)
['H', 'a', 'l']
>>> s3 = stop_if_double(iter([]))
>>> list(s3)
[]
```

(d) `sliding_window`; **2.5 Punkte**
Schreiben Sie eine Funktion `sliding_window`, die einen Iterator `it` und ein Zahl `n` als Argument nimmt und einen Generator zurückgibt, der Listen der Länge `n` aus `it` generiert. Dabei sollen die Listen jeweils um ein Element verschoben werden. Sind nicht genügend Elemente vorhanden, so soll ein leerer Generator zurückgegeben werden.

```
>>> s1 = sliding_window(iter(range(5)), 3)
>>> list(s1)
[[0, 1, 2], [1, 2, 3], [2, 3, 4]]
>>> s2 = sliding_window(iter("Hallo Welt :)"), 19)
>>> list(s2)
[]
```

Aufgabe 12.2 (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*[2], wenn jedes Element der Wertemengen des Dictionaries 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}}
>>> is_graph(example)
False
>>> example_graph = example | {3: set()}
>>> is_graph(example_graph)
True
>>> is_graph({"a": {"a", "aa"}})
False
>>> is_graph({})
True
```

(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 ein neues Dictionary `d` zurückgibt, das zu einem Graph ergänzt wurde. Fügen Sie dazu jeden Wert einer Wertemenge von `d`, der kein Schlüssel von `d` ist, als Schlüssel mit leerer Wertemenge in das Resultat ein.

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

(c) `nodes`, `edges`; **2.5 Punkte**

[2] 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)

Die Schlüssel in einem Graphen nennen wir *Knoten*. Jedes Tupel von Knoten `(a, b)`, bei dem `b` ein Element der Wertemenge 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 eine zweite Funktion `edges`, die ebenso einen Graph `graph` als Argument nimmt und einen Generator zurückgibt, der alle Kanten von `graph` produziert.

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

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

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

(e) `has_cycle`; **0 Punkte** (Knobelaufgabe, **schwer**)
Einen Zyklus der Länge `n>1` 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.

```
>>> has_cycle(example_graph)
True
>>> example_graph2 = {
...     0: {1}, 1: {2}, 2: set(), 3: {0},
...     4: {1, 5}, 5: {6}, 6: {7}, 7: {3, 8},
...     8: set(), 9: {8}
... }
>>> has_cycle(example_graph2)
False
>>> has_cycle(example_graph2 | {3: {0, 4}})
True
>>> has_cycle({"a": {"aa", "a"}, "aa": set()})
True
>>> has_cycle({1: {3}, 2: {1, 4}, 3: {4}, 4: set()})
False
>>> has_cycle({})
False
```

Aufgabe 12.3 (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 Webplattform. Halten Sie sich an das dort vorgegebene Format, da wir den Zeitbedarf mit einem Python-Skript automatisch statistisch auswerten. Die Zeitangabe `7.5 h` steht dabei für 7 Stunden 30 Minuten.