Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Marius Weidner, Hannes Saffrich
Simon Dorer, Sebastian Klähn

Universität Freiburg
Institut für Informatik
Wintersemester 2024

Übungsblatt 11

Abgabe: Montag, 13.01.2024, 9:00 Uhr

Aufgabe 11.1 (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) `fib`; **2.5 Punkte**

Schreiben Sie eine Funktion `fib`, die einen Generator zurückgibt, der die Elemente der Fibonacci-Folge generiert. Die Fibonacci-Folge ist eine unendliche Reihe von natürlichen Zahlen, die wie folgt definiert ist:

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_{n+2} &= f_{n+1} + f_n \quad \text{für } n \in \mathbb{N}_0 \end{aligned}$$

Die ersten Werte der Folge sind $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$.

Zum Beispiel:

```
>>> f = fib()
>>> fibs = []
>>> for _ in range(15):
...     fibs.append(next(f))
...
>>> fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

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

Schreiben Sie eine Funktion `random`, die einen Generator zurückgibt, der Pseudozufallszahlen[1] generiert. Die Funktion soll die vier ganzzahlige Argumente `seed`, `a`, `b` und `m` haben und einen Generator zurückgeben, der nacheinander die Ganzzahlen y_i der (unendlichen) Folge

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

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

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

```
>>> r = random(11, 5, 3, 64)
>>> rands = []
>>> for _ in range(15):
...     rands.append(next(r))
...
>>> rands
[58, 37, 60, 47, 46, 41, 16, 19, 34, 45, 36, 55, 22, 49, 56]
```

(c) `stop_if`; **2.5 Punkte**
Schreiben Sie eine Funktion `stop_if`, die einen Iterator `it` und ein Element `el` als Argumente nimmt und einen Generator zurückgibt, der solange Elemente aus `it` generiert, bis das `el` ausgegeben wird.

```
>>> s1 = stop_if(iter(range(5)), 3)
>>> list(s1)
[0, 1, 2, 3]
>>> s2 = stop_if(iter("Hallo Welt :)"), "X")
>>> list(s2)
['H', 'a', 'l', 'l', 'o', ' ', 'W', 'e', 'l', 't', ' ', ':', ')']
```

(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 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*[2], wenn jedes Element der Wertemengen des Dictionaries auch ein Schlüssel im selben Dictionary ist.

[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)

(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**
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. 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!

```
>>> 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 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 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.