Informatik I: Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Hannes Saffrich, Simon Ging
Wintersemester 2021

Universität Freiburg
Institut für Informatik

Übungsblatt 12

Abgabe: Montag, 24.01.2022, 9:00 Uhr morgens

Hinweis: Es gelten die selben Regeln wie bisher, diese können in Blatt 8 eingesehen werden.

Hinweis: Eine Generator-Funktion, die einen Generator von ganzen Zahlen als Argument nimmt und Strings generiert, hat folgende Typsignatur:

```
from typing import Iterator

def ints_to_strs(int_gen: Iterator[int]) -> Iterator[str]:
    for i in int_gen:
        yield str(i)
```

Hinweis: Attribute von Datenklassen können entweder als Argument bei Instanziierung mitgegeben werden oder automatisch nach der Initialisierung gesetzt werden. Weiterhin können Attribute optional sein. Betrachten Sie den Folgenden Code:

```
from typing import Optional
from dataclasses import dataclass, field


@dataclass
class Example:
    a: float
    b: float
    c: float = field(init=False)
    d: Optional[float] = None

    def __post_init__(self):
        self.c = self.a + self.b

example1 = Example(0.5, 1.5, d=7.0)
print(example1.c)
```

Die Ausgabe ist `2.0`. Attribute `a` und `b` müssen bei der Instanziierung mitgegeben werden, Attribut `c` wird automatisch erstellt. Die explizite Definition von `c` mit dem Ausdruck `c: float = field(init=False)` ist nicht zwingend notwendig und dient u.A. dazu, das Feld in die Feldliste und damit in die Repräsentation `str(example1)` aufzunehmen. Attribut `d` kann optional bei Instanziierung mitgegeben werden oder auf dem Standardwert belassen werden.

Hinweis: In diesem Blatt sollen Sie bestimmte Generatorfunktionen implementieren. Diese sollen dabei das folgende Kriterium erfüllen: Um das nächste Element zu generieren, dürfen nur die Berechnungen durchgeführt werden, die dafür auch strikt notwendig sind. Insbesondere dürfen die Berechnungen nicht durchgeführt werden, welche die darauffolgenden Elemente erzeugen.[1] **Nur** für die Ausgabe des Ergebnisses dürfen Sie `list` auf einen Generator anwenden. Sie dürfen **nicht** `itertools` verwenden.

Aufgabe 12.1 (Generatoren; Datei: `generators.py`; Punkte: 5)

(a) Implementieren Sie die Generatorfunktion `collatz` mit Argument `n` (`int`) wie folgt:

$c_0 = \texttt{n}$

$$c_{n+1} = \begin{cases} c_n/2 & \text{if } x \bmod 2 = 0 \\ 3c_n + 1 & \text{otherwise} \end{cases}$$

```
>>> generator = collatz(11)
>>> for i in range(20):
>>>     print(next(generator), end=" ")
>>> print()
11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 4 2 1 4 2
```

(b) Implementieren Sie die Generatorfunktion `generate_target`. Diese nimmt einen Generator `generator` und einen Wert `target` (`int`) als Argument und generiert die Werte des Generators, bis der Wert `target` generiert wurde.

```
>>> print(list(generate_target(iter(range(10)), 4)))
[0, 1, 2, 3, 4]
>>> print(list(generate_target(collatz(11), 1)))
[11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

(c) Implementieren Sie die Generatorfunktion `arithmetic_mean`. Diese nimmt einen Generator `generator` als Argument und generiert den arithmetischen Mittelwert der Werte des Generators. Insbesondere dürfen Sie nicht die einzelnen generierten Werte in einer Liste speichern.

```
>>> print(list(arithmetic_mean(iter(range(0, 21, 4)))))
[0.0, 2.0, 4.0, 6.0, 8.0, 10.0]
```

[1] Der Sinn eines Generators besteht ja gerade darin, die Elemente erst bei Bedarf zu generieren ("lazy evaluation"). Dadurch kann in bestimmten Fällen Speicherbedarf und Ausführungszeit gespart werden. Andernfalls könnten wir auch gleich eine normale Funktion schreiben, die eine Liste aller zu generierenden Elemente zurückgibt (zumindest wenn der Generator endlich ist).

(d) Verwenden Sie die builtin-Funktion `map` und eine von Ihnen zu erstellende Hilfsfunktion `map_helper` (Argument `x` (`int`), Rückgabe `int`), um die Werte der gegebene Range modulo 7 zu berechnen.

Eingabedaten:

```
>>> input_generator = iter(range(0, 26, 5))
```

Erforderliche Ausgabe Ihres Codes:

```
[0, 5, 3, 1, 6, 4]
```

(e) Verwenden Sie die builtin-Funktion `filter` und eine von Ihnen zu erstellende Hilfsfunktion `filter_helper` (Argument `x` (`int`), Rückgabe `bool`), um von der gegebenen Range nur die Werte auszugeben, welche durch 3 oder 5 teilbar sind. Eingabedaten:

```
>>> input_generator = iter(range(20))
```

Erforderliche Ausgabe Ihres Codes:

```
[0, 3, 5, 6, 9, 10, 12, 15, 18]
```

Aufgabe 12.2 (CachedTuple; Datei: `cachedtuple.py`; Punkte: 5)

Wiederholung: Ein `Iterable` definiert die Funktion `__iter__`, welche einen `Iterator` zurückgibt. Ein `Iterator` ist ein `Iterable` und definiert zusätzlich die Funktion `__next__`, welche das nächste Element des Iterators zurückgibt. Ein `Generator` ist ein spezieller `Iterator`.

Ein Nachteil von Generatoren ist, dass nicht ohne Weiteres ein Wert mit einem bestimmten Index zurückgegeben werden kann. Hierfür könnte der Generator in eine Liste umgewandelt werden, dies ist aber nicht möglich für unendliche Generatoren und ist unter Umständen ineffizient.

Um dieses Problem zu lösen, erstellen Sie die Datenklasse `CachedTuple`. Das Ziel der Klasse ist es, ein `Iterable` effizient auszuwerten. Wenn der Wert mit index `i` angefragt wird, soll das `Iterable` bis zu Wert `i` durchlaufen werden, alle Werte zwischengespeichert werden und dann der Wert mit Index `i` zurückgegeben werden.

Das `Iterable` soll innerhalb eines solchen Objekts nur ein einziges Mal durchlaufen werden.

Definieren Sie die folgenden Attribute:

- Ein Feld `iterable` vom Typ `Iterable`.
- Ein optionales Feld `n_max` vom Typ `int`.

- Ein Attribut `iterator` vom Typ `Iterator`, welches automatisch einen Iterator aus dem Feld `iterable` erzeugt.
- Eine Attribut `cache`, automatisch initialisiert als eine leere Liste.
- Ein Attribut `finished`, automatisch initialisiert als ein `bool` mit Wert `False`. Dieses Attribut soll angeben, ob der Iterator bereits vollständig im Cache liegt.

Definieren Sie die folgenden Methoden:

- `cache_next`: Fügt das nächste Element des Iterators `iterator` in den Cache ein, falls der Iterator noch nicht fertig durchlaufen ist und das optionale Argument `n_max` nicht überschritten wird. Fangen Sie ggf. das Ende des Iterators mit `try...except` ab.
- `__getitem__` mit Attribut `item`, Typ `int`. Diese Methode wird aufgerufen, wenn ein Objekt mit rechteckigen Klammern indiziert wird, beispielweise ruft `obj[7]` die Methode `__getitem__` mit Argument `item = 7` auf. Sie kennen diese Aufrufe von der Indizierung von Listen oder Dictionaries. Prüfen Sie mit Pattern Matching, ob `item` ein `int` ist, andernfalls lösen Sie einen `TypeError` aus. Es soll das Element mit Nummer `item` zurückgegeben werden. Falls `item` negativ ist, geben Sie einen `IndexError` aus. Falls das Element bereits im `cache` liegt, geben Sie es zurück. Solange das Element noch nicht im `cache` liegt, füllen Sie den `cache` mit `cache_next`. Falls die Nummer des Elements größer als die maximale Größe des Caches `n_max` ist oder keine Elemente mehr im `iterator` sind, lösen Sie einen `IndexError` aus.
- `__len__`: Gibt die Länge des Iterators zurück. Hierfür muss der gesamte Iterator durchlaufen werden, stellen Sie also sicher, dass Sie diese Funktion nicht unnötig innerhalb der Klasse aufrufen.

Hinweis: Die Leseposition ist unabhängig von der Länge des Caches.

Beispielaufruf:

```
>>> iterable = range(20)
>>> cachedtuple = CachedTuple(iterable)
>>> print(cachedtuple[0])
0
>>> print(len(cachedtuple.cache))
1
>>> print(cachedtuple[10])
10
>>> print(len(cachedtuple.cache))
11
>>> print(len(cachedtuple))
20
>>> print(len(cachedtuple.cache))
20
>>> print(cachedtuple[25])
```

```
Traceback ...
IndexError: Index 25 out of range
```

Durch die Definition von `__getitem__` mit akzeptierten `int` ab 0 können automatisch `iter`, `list` und `for loops` darauf angewendet werden.

```
>>> print(list(cachedtuple))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> print(next(iter(cachedtuple)))
0
>>> for x in cachedtuple:
>>>     print(x, end=" ")
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Verwenden Sie die Datei `test_cachedtuple.py` um Ihre Implementierung zu testen.

Aufgabe 12.3 (Pfadfinder; Datei: `pathfinder.py`; Punkte: $8 = 2 + 4 + 2$)

Gegeben ist ein 2D-Labyrinth, welches aus Wänden und freien Feldern besteht. Es sollen alle möglichen Pfade vom Start zum Ziel sowie der kürzeste Pfad gefunden werden. Das Labyrinth ist wie folgt definiert, wobei 1 ein freies Feld und 0 ein Hindernis angibt:

```
world = [[1, 1, 1, 0],
     [1, 1, 1, 0],
     [1, 0, 1, 1],
     [1, 0, 1, 1]]
```

Der Start ist oben links und das Ziel ist unten rechts. Ein Pfad ist definiert als Liste aus 2-Tuplen, jedes Tupel beinhaltet die x- und y-Koordinate, wobei x die Spalte und y die Zeile angibt. Alle Koordinaten sind ganze Zahlen ≥ 0, wobei die erste Zeile und Spalte die Koordinate 0 haben.

(a) Erstellen Sie die Funktion `visualize_path`. Diese nimmt die Parameter `world` und `path` wie oben beschrieben und visualisiert das Labyrinth: "#" für Wände, "." für freie Felder und "X" für einzelne Schritte auf dem Pfad. Beispielaufruf:

```
>>> visualize_path(world, [(0,0), (1, 0)])
XX.#
...#
.#..
.#..
```

(b) Schreiben Sie die Generatorfunktion `find_paths`. Diese soll rekursiv alle möglichen Pfade zwischen Start und Ziel ablaufen und alle erfolgreichen Pfade generieren. Ein Pfad ist erfolgreich, wenn

- er beim Start beginnt und beim Ziel endet,
- keine Wände betreten werden,
- kein Feld außerhalb des Spielfelds betreten wird und
- kein Feld zweimal betreten wird.

Die Generatorfunktion soll als Argumente die Parameter `world`, `sx` und `sy` für die Startposition, `ex` und `ey` für die Zielposition und `path` für den bisher gelaufenen Pfad haben.

```
>>> print(next(find_paths(world, 0, 0, 3, 3, [])))
[(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (3, 3)]
```

Weitere Hinweise:

- Verwenden Sie backtracking wie in der Vorlesung beschrieben.
- Laufen Sie die umliegenden Felder in der Reihenfolge oben, rechts, unten, links ab.
- Listen sind veränderliche Objekte (mutable), Sie müssen also darauf achten, den bisher gelaufenen Pfad beim Backtracking nicht zu zerstören. Eine Lösungsmöglichkeit: Sie kopieren den Pfad bei jedem neuen Schritt. Eine andere Möglichkeit: Sie machen die Schritte beim Backtracking jeweils rückgängig und geben am Ende eine Kopie des vollständigen Pfades zurück. Sie können flache Listen mit `copy` und geschachtelte Listen mit `deepcopy` kopieren (beide aus dem Modul `copy`) oder alternativ ein neues Listenobjekt erstellen.
- Der Vorteil der Generatorfunktion darf nicht verloren gehen, sammeln Sie also nicht alle Pfade in einer Liste auf. Es muss möglich sein, nur den ersten Pfad zu bekommen und dann abzubrechen.
- Verwenden Sie `yield from` für den rekursiven Aufruf der Generatorfunktion.

(c) Schreiben Sie die Funktion `get_shortest_path`. Diese nimmt als Argument einen Generator `path_generator`, welcher mit `find_paths` erstellt wurde. Die Funktion soll die Anzahl der Pfade und die Anzahl Schritte des kürzesten Pfades in die Konsole schreiben und anschließend den kürzesten Pfad zurückgeben:

```
>>> shortest_path = get_shortest_path(find_paths(world, 0, 0, 3, 3, []))
8 Pfade gefunden, der kürzeste Pfad hat 7 Schritte.
>>> visualize_path(world, shortest_path)
XXX#
..X#
.#XX
.#.X
```

Verwenden Sie die Datei `test_pathfinder.py`, um Ihre Implementierung zu testen.

Abschließende Bemerkung: Der Algorithmus arbeitet mit "Brute Force". Für praktische Anwendungen wäre eine solche Umsetzung zum Finden von Pfaden üblicherweise zu langsam.

Aufgabe 12.4 (Erfahrungen; 2 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 `7.5 h` steht dabei für 7 Stunden 30 Minuten.