PSPP-Praktikum 14
Python (Ergänzungen, keine Abgabe)

Die folgenden Aufgaben setzen Kenntnisse zum Einsatz von Klassen in Python voraus, ein Thema, das in diesem PSPP-Durchgang nicht prüfungsrelevant ist. Für Interessierte sind die Aufgaben in diesem Übungsblatt zusammengestellt.

1. Lazy Chain

(fakultativ, zur Vertiefung)

Wenn Sie schon in JavaScript entwickelt haben, kennen Sie vermutlich das Verketten von Methodenaufrufen (Chaining) in jQuery. In der Bibliothek Underscore.js gibt es eine spezielle Funktion *chain*, mit der eine solche Aufrufkette initiiert werden kann. Auf diese Weise ist Code in der Regel übersichtlicher als mit einer Reihe von verschachtelten Aufrufen. Aus der Dokumentation:

_.chain(obj) – Returns a wrapped object. Calling methods on this object will continue to return wrapped objects until *value* is called.

Als Beispiel ist angegeben:

```
var stooges = [{name:'curly', age:25}, {name:'moe', age:21}, {name:'larry', age:23}];
var youngest = _.chain(stooges)
  .sortBy(function(stooge){ return stooge.age; })
  .map(function(stooge){ return stooge.name + ' is ' + stooge.age; })
  .first()
  .value();
=> "moe is 21"
```

Hier werden alle verketteten Operationen ausgeführt, auch wenn der Wert am Ende nicht mit *value* extrahiert wird. Die folgende Python-Klasse implementiert eine *LazyChain*, das heisst die Funktionen werden vorbereitet, aber erst am Ende durch die *run*-Methode ausgeführt:

```
class LazyChain:

    def __init__(self, obj):
        self.calls = []
        self.target = obj

    def do(self, methodname, *args):
        def methodcall(target):
            method = getattr(target, methodname)
            return method(*args)

        self.calls.append(methodcall)
        return self

    def run(self):
        from functools import reduce
        return reduce(lambda target,fun: fun(target), self.calls, self.target)
```

Vermutlich ist der Aufbau der Klasse nicht gleich auf Anhieb zu verstehen. Probieren wir ein Beispiel. Die *LazyChain* wird hier mit einer Menge {1, 2, 3} initialisiert:

```
>>> LazyChain({1,2,3}) \
...     .do("union",{6,7,8}) \
...     .do("difference",{2,7}) \
...     .run()
{8, 1, 3, 6}
```

Aufgaben:

- Der Konstruktor initialisiert einen Stack von Aufrufen, die Methode *do* fügt diesem Stack einen Aufruf hinzu. Was genau wird auf dem Stack abgelegt? Das *getattr* liefert die Methode des Objekts *target* mit dem Namen *methodname*.

- Die Methode *run* arbeitet die Methode mit Hilfe von *reduce* ab. Auch diese Zeile hat es in sich. Erklärungshilfe: Es wird die Liste der Aufrufe abgearbeitet. Startwert ist das Ausgangsobjekt *self.target*, im Beispiel die Menge {1, 2, 3}. Versuchen Sie sich klar zu machen, warum das *reduce* für eine Liste [fun1, fun2, fun3] in *self.calls* die folgenden Aufrufe ausführt:

  ```
  fun3(fun2(fun1(self.target)))
  ```

- Erweitern Sie die Klasse *LazyChain* um eine Methode *tap*, die es ermöglicht, den aktuellen Wert des Target-Objekts zu verarbeiten (zum Beispiel auszugeben). Der Methode *tap* wird eine Funktion übergeben, die dann mit dem Target-Objekt aufgerufen wird. Sie ruft zwar die Funktion auf, gibt aber das Objekt selbst unverändert zurück.

```
>>> LazyChain({1,2,3}) \
...     .do("union",{6,7,8}) \
...     .tap(print) \
...     .do("difference",{2,7}) \
...     .run()
{1, 2, 3, 6, 7, 8}
{8, 1, 3, 6}
```

Auch die *print*-Anweisung des *tap*-Aufrufs wird erst nach Aufruf von *run* ausgeführt:

```
>>> delayed_activity = LazyChain({1,2,3}) \
...     .do("union",{6,7,8}) \
...     .tap(print) \
...     .do("difference",{2,7})
>>> delayed_activity
<__main__.LazyChain object at 0x10c2825f8>
>>> delayed_activity.run()
{1, 2, 3, 6, 7, 8}
{8, 1, 3, 6}
```

Hinweis: Als Beispielobjekt wurde hier eine Menge gewählt. Der Vorteil ist hier, dass die Mengenoperationen wie *union* und *difference* den neuen Wert zurückliefern. Das ist bei vielen Listenoperationen nicht der Fall: *sort* und *extend* beispielsweise ändern die Liste an Ort und Stelle, ohne den neuen Wert zurückzugeben.

Quellenangaben:

Einige Ideen (allerdings in JavaScript umgesetzt) stammen aus:

Michael Fogus: Functional JavaScript, Introducing Functional Programming with Underscore.js, O'Reilly, 2013

2. Klassen: Queue

(fakultativ, zur Vertiefung)

Queues gibt es natürlich bereits in der Python-Library. Wir wollen hier aber unsere eigene Implementierung nach eigenen Anforderungen aufbauen.

Implementieren Sie eine Klasse, die eine Queue von Elementen repräsentiert. Bei einer Queue werden Elemente immer an einem Ende eingefügt und am anderen entfernt. Die Elemente werden also in der gleichen Reihenfolge aus der Queue entnommen, in der sie in der Queue abgelegt wurden (first in, first out: FIFO; im Gegensatz zu einem Stack, dort gilt last in, first out: LIFO).

Die Queue soll als Liste implementiert werden. Dadurch kann sie beliebige Datentypen aufnehmen. Implementieren Sie einen Konstruktor, der eine leere Queue anlegt. Eine Methode *add(elem)* soll Elemente in die Queue einfügen und eine Methode *get()* soll Elemente aus der Queue entfernen.

Ein Problem kann auftreten, wenn die Methode *get()* aufgerufen wird, obwohl die Queue leer ist. In dieser ersten Version soll *get()* in diesem Fall einfach ein "*" zurückgeben.

Test mit Pizza

(Hinweis: Wenn Sie schon zu viele Pizza-Queues geschrieben haben, können Sie auch irgendeine andere Warteschlange simulieren).

Um die Queue zu testen, soll nun ein Pizza-Service simuliert werden. Alle Bestellungen werden in einer Queue abgelegt und in der Reihenfolge des Eingangs abgearbeitet. In einer Schleife (mit zum Beispiel 50 Durchgängen) wird zunächst per Zufall entschieden, ob eine Bestellung eingeht oder die nächste Pizza in der Warteschlange hergestellt werden soll. Falls eine Bestellung eingeht, wird ebenfalls mit dem Zufallszahlengenerator entschieden, welche Pizza aus einer gegebenen Auswahl bestellt wurde und in die Queue einzutragen ist.

Die Ausgabe des Scripts könnte ungefähr folgendermassen aussehen:

```
-> Pizza herstellen: *
-> Pizza herstellen: *
Neue Bestellung: Sole Mio
Neue Bestellung: Sole Mio
Neue Bestellung: Speziale
-> Pizza herstellen: Sole Mio
Neue Bestellung: Prosciutto
-> Pizza herstellen: Sole Mio
Neue Bestellung: Prosciutto
-> Pizza herstellen: Speziale
```

Queue mit Exceptions

Die Rückgabe von "*" der *get()*-Methode als Hinweis auf eine leere Queue ist noch nicht optimal. Ändern Sie die Methode so ab, dass eine Exception ausgelöst wird, wenn die *get()*-Methode bei leerer Queue aufgerufen wird. Am besten definieren Sie dazu eine (leere) Exception-Klasse:

```
class MyQueueError(Exception):
    pass
```

Der Aufruf der *get()*-Methode erfolgt dann in einer *try/except*-Anweisung. Im Exception-Handler kann dann anstelle der Ausgabe „Pizza herstellen: *“ einfach „Pause...“ ausgegeben werden.

Queue mit Index-Operator

Ergänzen Sie Ihre Queue-Implementierung um den Index-Operator [ ]. Dadurch kann die Queue zum Beispiel in *for... in...* –Anweisungen verwendet werden.