Ein abstrakter Datentyp: Die Warteschlange

Abstrakte Datentypen [ADT] sind ein Konzept zur Beschreibung einer Datensammlung im Zusammenhang mit den auf ihnen arbeitenden Operationen (Funktionen oder Prozeduren, in der OO Methoden). In objektorientierten Programmiersprachen realisiert man sie in Klassen, da diese – wie von der Konzeption der ADT gefordert – nicht nur die Möglichkeit bieten, Daten und Funktionen zu koppeln, sondern sie auch zu kapseln.

Warteschlange als Beispiel

Eine Warteschlange [englisch queue] enthält beliebige Objekte, für die man – orientiert an Warteschlangen beim Arzt oder an einem Schalter – die Funktionalität durch die folgenden Sätze beschreiben kann:

- Bitte hinten anstellen!
- Der Nächste bitte!
- Wer ist der Nächste?
- Wie viele warten schon?
- Keiner da?
- Wer ist der Letzte?

In diesem Text gehe ich nicht auf die üblicherweise verwendeten englischen Namen für die Methoden ein, sondern verwende:

- **`Anstellen`**
- **`Aufrufen`**
- **`DerNaechsteIst`**
- **`GibLaenge`**
- **`Leer`**
- **`Letzter`**

Formulierung der Aufgaben der Methoden:

Dazu verwende ich die Ausdrücke Kopf und Schwanz für den Anfang und das Ende der Warteschlange.

- **`Anstellen`** hängt ein neues Element am Ende der Warteschlange an.
- **`Aufrufen`** setzt den Kopf der Warteschlange weiter und gibt den Inhalt des bisherigen Kopfes zurück.
- **`DerNaechsteIst`** gibt dagegen allein den Inhalt am Kopf der Warteschlange zurück, ohne sie weiter zu setzen.
- **`GibLaenge`** gibt die aktuelle Länge der Warteschlange zurück.
- **`Leer`** gibt zurück, ob die Warteschlange leer ist.
- **`Letzter`** gibt das letzte Element der Warteschlange zurück.

Kapselung der Implementierung

Zum Grundkonzept eines ADT gehört die Kapselung der Implementierung. Er lässt sich in Java also einfach durch ein Interface beschreiben, das der Javadoc im Interface Queue zu entnehmen ist. Definiert ist die Schnittstelle, also wie mit einer konkreten Queue interagiert werden kann, es ist aber keine Implementation daran gebunden.
In Python könnte man zunächst eine abstrakte Basisklasse (abc) realisieren. Sie ist dann mit einem Java-Interface vergleichbar, da in ihr alle Methoden nur abstrakt definiert werden können und man jede davon erbende konkrete Klasse zwingt, diese zu implementieren. Ein Beispiel finden Sie auf der letzten Seite des Textes.

Einfach verkettete Liste

In einer Implementation werden wir eine einfach verkettete Liste verwenden, wie sie von Lisp/Scheme verwendet wird. Dort wird ein Listenelement intern mit einem Paar von Zeigern[1] realisiert, dessen erste (car genannt) auf den Inhalt und dessen zweiter (cdr genannt) auf ein nachfolgendes Listenelement zeigen.

Eine Klasse für Listenelemente zu unserer Warteschlange

```
class Element_WS(object) :
    '''Klasse für ein Glied einer Warteschlange
    Der Inhalt kann nachträglich nicht gesetzt werden.'''
    def __init__(self, inhalt) :
        self.__inhalt = inhalt      # object
        self.__naechstes = None     # Element_WS

    def SetzeNachfolger (self, nachfolger) :
        self.__naechstes = nachfolger

    def GibInhalt (self) :
        return self.__inhalt

    def GibNachfolger (self) :
        return self.__naechstes
```

Zwingend sind die angegebenen Methoden. Man könnte noch eine Set-Methode für den Inhalt hinzufügen. Von unserer Anwendung her gibt es dafür aber keinen Anlass. Im Gegenteil: Es sollte gewährleistet sein, dass die Inhalte der Elemente nicht verändert werden können. Der Nachfolger dagegen muss gesetzt werden können, da die Elemente einer Warteschlange stets am Ende angehängt werden und daher zunächst kein Nachfolger existiert. Der Nachfolgezeiger muss zwingend mit None belegt werden.

Die Klasse Warteschlange zu diesem Datenmodell

Im Konstruktor werden Kopf und Schwanz zunächst auf None gesetzt:

```
    def __init__(self) :
        '''Konstruktor'''
        self.__kopf = None
        self.__schwanz = None
```

Bei der Methode zum Anstellen ist zu beachten, dass die Warteschlange am Anfang leer ist. Das neu erzeugte Element bildet in dem Fall gleichzeitig Kopf und Schwanz der Warteschlange.

```
    def Anstellen (self, objekt) :
        '''hängt ein neues Element am Ende der Warteschlange an'''
        neu = Element_WS(objekt)
        if self.__kopf == None:
            self.__kopf = neu
            self.__schwanz = neu
        else:
            self.__schwanz.SetzeNachfolger(neu)
            self.__schwanz = neu
```

1 Zeiger sind Adressen im Speicher, unter denen die Werte abgelegt sind.

Im Normalfall wird der Nachfolger des aktuell letzten Elementes das neue und dann das neue als aktuelles letztes Element gesetzt.
Ähnliche Fallunterscheidungen sind beim Aufrufen notwendig:

```
def Aufrufen (self) :
    '''setzt den Kopf der Warteschlange weiter und
    gibt den Inhalt des bisherigen Kopfes zurück'''
    if self.__kopf == None:
        return None
    kopfObjekt = self.DerNaechsteIst()
    self.__kopf = self.__kopf.GibNachfolger()
    if self.__kopf == None:
        self.__schwanz = None
    return kopfObjekt
```

Wurde das einzige verbliebene Element der Warteschlange aufgerufen, der Kopf wird dann None, muss der Schwanz ebenfalls auf None gesetzt werden. Wie bei der Abfrage des ersten Elementes muss bei einer leeren Warteschlange None zurückgegeben werden, alternativ könnte man auch eine Fehlermeldung ausgeben.

```
def DerNaechsteIst (self) :
    '''gibt den Inhalt am Kopf der Warteschlange zurück'''
    if self.__kopf == None:
        return None
    return self.__kopf.GibInhalt()
```

Damit lässt sich einfach die Bedingung für die Abfrage, ob die Warteschlange leer ist formulieren:

```
def IstLeer (self) :
    '''gibt zurück, ob die Warteschlange leer ist'''
    return (self.__kopf == None)
```

Iterator

Für die noch fehlende Methode **`GibLaenge`** ist es hilfreich, einen Iterator zu nutzen. Ein Iterator muss auch implementiert werden, wenn man für die Warteschlange die for-Schleife einsetzen will.
Der Iterator benötigt zwei Methoden in der Klasse Warteschlange,

- die besondere Methode **`__iter__`** und
- die Methode **`__next__`** .

```
def __iter__(self):
    '''initialisiert den Iterator'''
    self.aktuell=self.__kopf
    return self
```

Die Methode entspricht einem Konstruktor für den Iterator. Das Attribut aktuell wird als Zeiger auf das aktuelle Element genutzt und muss daher am Beginn auf den Kopf gesetzt werden.
Das Problem, mit einer leeren Warteschlange umzugehen, muss in der Methode **`next`** gelöst werden, deren Hauptaufgaben das Weitersetzen des Zeigers für die Iteration und die Rückgabe des aktuellen Wertes sind.

```
def __next__(self):
    '''definiert den Iterationsschritt'''
    if self.aktuell==None:
        raise StopIteration
    else:
        wert=self.aktuell.GibInhalt()
        self.aktuell=self.aktuell.GibNachfolger()
        return wert
```

Im Abbruchfall – standardmäßig also am Ende der Warteschlange – muss die Methode eine Exception vom Typ **`StopIteration`** auslösen, die von der **`for`**-Schleife benötigt wird.

GibLaenge

Die Methode kann nun den Iterator bei der Berechnung der Anzahl der Elemente nutzen:

```
def GibLaenge (self) :
    '''gibt die aktuelle Länge der Warteschlange zurück'''
    laenge = 0
    for element in self:
        laenge += 1
    return laenge
```

Warteschlange intern mit der Standardliste implementieren

Natürlich kann man intern eine normale Liste verwenden.

Aufgabe:

- Schreiben Sie eine Klasse Warteschlange mit einer normalen Liste zur internen Verwaltung der Daten.
- Prüfen Sie, ob die Klasse weiterhin dieselbe Schnittstelle hat wie die oben beschriebene.
- Begründen Sie, weshalb sich die Schnittstelle nicht ändern sollte.

Anhang 1: Eine Testklasse

```
class TestWarteschlange(unittest.TestCase):

    def setUp(self):
        self.warteschlange = Warteschlange()

    def test_Initialisierung(self):
        self.assertTrue(self.warteschlange.IstLeer())
        self.assertEqual(self.warteschlange.GibLaenge(), 0)
        self.assertEqual(self.warteschlange.DerNaechsteIst(), None)
        self.assertEqual(self.warteschlange.Letzter(), None)
        # Problem Aufrufen bei leerer Warteschlange
        self.assertEqual(self.warteschlange.Aufrufen(), None)

    def test_Aktion(self):
        self.warteschlange.Anstellen('Hund')
        self.assertFalse(self.warteschlange.IstLeer())
        self.assertTrue(self.warteschlange.GibLaenge()==1)
        self.assertEqual(self.warteschlange.DerNaechsteIst(), 'Hund')
        self.assertEqual(self.warteschlange.Letzter(), 'Hund')

        self.warteschlange.Anstellen('Katze')
        self.assertFalse(self.warteschlange.IstLeer())
        self.assertTrue(self.warteschlange.GibLaenge()==2)
        self.assertEqual(self.warteschlange.DerNaechsteIst(), 'Hund')
        self.assertEqual(self.warteschlange.Letzter(), 'Katze')

        self.warteschlange.Anstellen('Vogel')
        self.assertFalse(self.warteschlange.IstLeer())
        self.assertTrue(self.warteschlange.GibLaenge()==3)
        self.assertEqual(self.warteschlange.DerNaechsteIst(), 'Hund')
        self.assertEqual(self.warteschlange.Letzter(), 'Vogel')

        self.assertEqual(self.warteschlange.Aufrufen(), 'Hund')
        self.assertFalse(self.warteschlange.IstLeer())
        self.assertTrue(self.warteschlange.GibLaenge()==2)
        self.assertEqual(self.warteschlange.DerNaechsteIst(), 'Katze')
        self.assertEqual(self.warteschlange.Letzter(), 'Vogel')

        self.assertEqual(self.warteschlange.Aufrufen(), 'Katze')
        self.assertFalse(self.warteschlange.IstLeer())
        self.assertTrue(self.warteschlange.GibLaenge()==1)
        self.assertEqual(self.warteschlange.DerNaechsteIst(), 'Vogel')
        self.assertEqual(self.warteschlange.Letzter(), 'Vogel')

        self.assertEqual(self.warteschlange.Aufrufen(), 'Vogel')
        self.assertTrue(self.warteschlange.IstLeer())
        self.assertTrue(self.warteschlange.GibLaenge()==0)
        self.assertEqual(self.warteschlange.DerNaechsteIst(), None)
        self.assertEqual(self.warteschlange.Letzter(), None)
        # Problem: Was liefert Aufrufen jetzt?
        self.assertEqual(self.warteschlange.Aufrufen(), None)

    def test_Iteration(self):
        self.warteschlange.Anstellen('Hund')
        self.warteschlange.Anstellen('Katze')
        self.warteschlange.Anstellen('Vogel')
```

```
        ausgabe = []
        for element in self.warteschlange:
            ausgabe.append(element)
        self.assertEqual(ausgabe, ['Hund', 'Katze', 'Vogel'])
```

Anhang 2: abstract-base-class statt Interface

```
from abc import * # abstrakte Klasse realisieren

class Warteschlange_abc(object) :
    '''Oberklasse für Warteschlangen'''
    __metaclass__ = ABCMeta

    def __init__(self) :
        '''Konstruktor: nichts zu tun'''
        pass

    @abstractmethod
    def __iter__(self):
        '''initialisiert den Iterator'''
        return NotImplemented

    @abstractmethod
    def next(self):
        '''definiert den Iterationsschritt. Muss überschrieben werden und
          im Abbruchfall "raise StopIteration" auslösen.'''
        return NotImplemented

    @abstractmethod
    def Anstellen (self, objekt) :
        '''hängt ein neues Element am Ende der Warteschlange an'''
        return NotImplemented

    @abstractmethod
    def Aufrufen (self) :
        '''setzt den Kopf der Warteschlange weiter und
         gibt den Inhalt des bisherigen Kopfes zurück'''
        return NotImplemented

    @abstractmethod
    def DerNaechsteIst (self) :
        '''gibt den Inhalt am Kopf der Warteschlange zurück'''
        return NotImplemented

    @abstractmethod
    def GibLaenge (self) :
        '''gibt die aktuelle Länge der Warteschlange zurück'''
        return NotImplemented

    @abstractmethod
    def IstLeer (self) :
        '''gibt zurück, ob die Warteschlange leer ist'''
        return NotImplemented

    @abstractmethod
    def Letzter(self):
        '''gibt das letzte Element der Warteschlange zurück'''
```

```
        return NotImplemented
```