LV 7281
Skriptsprachen-Praktikum
Übung 02

Ruby-Klassen und Methoden anlegen
Die Klasse Array

Organisatorisches

- Arbeitsverzeichnis:

```
~/lv/skriptspr/02/
```

- Dateinamen:

```
02-container.rb   # neu erstellen & abgeben
02-ctests.rb      # neu erstellen & abgeben
```

- Werkzeuge:

```
ruby              # Der Interpreter
```

- Vorlagen:

```
(keine)
```

Die Aufgabe

- Allgemeine Beschreibung
  - Erzeugen Sie zwei „Klassiker“ der Informatik als einfache Klassen auf der Basis der eingebauten Ruby-Klasse „Array“ bzw. der Python-Klasse „List”
  - Entwickeln Sie dazu passenden Testcode

- Material, Hinweise (Ruby):
  - Nutzen Sie Attribute zum Speichern der internen Werte ihrer Objekte.
  - Nutzen Sie das Delegationsprinzip – delegieren Sie die eigentliche Arbeit an Methoden der Klasse `Array`
  - Die Klasse `Array` ist sehr leistungsfähig. Kenntnis ihrer Methoden ist lohnend, nicht nur für diese Übung. Lesen Sie die Dokumentation!
  - Bitte NICHT die eingebauten Möglichkeiten verwenden (wie z.B. die Queue-Klasse aus dem Thread-Modul)

Die Aufgabe

A: Implementieren Sie eine Klasse **MyQueue** (FIFO) mit folgenden Methoden:

```
initialize(max_size=0)
  # Eine leere Warteschlange mit max_size Plätzen
  #  anlegen. max_size==0 : Keine Platzbegrenzung

enqueue(obj [, obj2, …])    --> anInteger
  # Objekt hinten in Warteschlange einreihen
  # Rückgabewert = Anzahl übernommener Objekte
  #    Normalfall: 0 (kein Platz mehr) oder 1
dequeue                     --> anObject
  # Objekt am vorderen Ende der Warteschlange entnehmen
  # und zurückliefern; nil falls Queue leer
peek                        --> anObject
  # Referenz auf Objekt am Ausgang (vorderen Ende) der
  # Warteschlange liefern, nil falls Queue leer
```

Die Aufgabe

B: Implementieren Sie eine Klasse **MyStack** (LIFO) mit folgenden Methoden:

```
initialize(max_size=0)
  # Einen leeren Stack mit max_size Fächern anlegen
  #  max_size==0 : Keine Platzbegrenzung

push(obj [, obj2, …])  --> anInteger
  # Objekt auf den Kellerspeicher legen
  # Rückgabewert = Anzahl übernommener Objekte
  #    Normalfall: 0 (kein Platz mehr) oder 1
pop                    --> anObject
  # Objekt vom Kellerspeicher nehmen und zurückgeben
  # nil falls Stack leer
peek                   --> anObject
  # Referenz auf oberstes Objekt des Stacks liefern,
  # nil falls Stack leer
depth                  --> anInteger
  # Anzahl gespeicherter Objekte. Nutzen Sie „alias“!
```

Die Aufgabe

C: Implementieren Sie die folgenden gemeinsame Methoden der Klassen **MyQueue** und **MyStack**:

```
length, size            --> anInteger
  # Anzahl gespeicherter Objekte. Nutzen Sie „alias“!

clear                   --> anInteger
  # Queue bzw. Stack löschen,
  # Anzahl Einträge vor dem Löschen zurückgeben.

empty?                  --> aBoolean
  # true, falls Queue bzw. Stack leer
```

Die Aufgabe

C: Schreiben Sie Test-Code zu beiden Klassen

Testen Sie möglichst viele Aspekte der Klassen

a) Testen Sie jede Methode.

b) Testen Sie insbesondere Grenzfälle wie leere Warteschlange oder Stack-Überlauf.

c) Ausgabe einer Test-Statistik?

HINWEISE

- Der Nutz-Code soll in der Datei „02-container.rb“ stehen.
- Der Test-Code wird in die Datei „02-ctests.rb“ geschrieben.
- Einbinden mit **`require "./02-container"`** (ohne .rb)
- Achtung: Der Dozent wird Ihren Nutz-Code mit seinen eigenen Unit-Tests automatisiert prüfen!

(*) Ausbaustufe 1: Redundanzen vermeiden mittels einer gemeinsamen Basisklasse „Container“

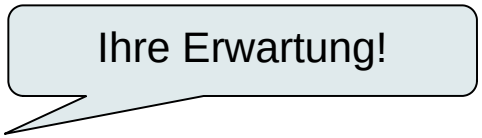
(*) Ausbaustufe 2: enqueue() und push() akzeptieren auch eine Liste von Objekten.

Bem.: (*) heißt hier: Optionaler Aufgabenteil

Kleine Hilfe

Hilfestellung: Etwas Source-Code als Ausgangspunkt

```
# Nutz-Code in 02-container.rb:

class MyQueue
  def initialize(max_size=0)
    @buf = []
    @max_size = max_size
  end


  def size
    @buf.size   # Arbeit an @buf delegieren…
  end
end


# Test-Code in 02-ctests.rb:

require "./02-container"


q = MyQueue.new(5)
puts "Fehler 1" unless q.size == 0
```



Kleine Hilfe

Für **JUnit**-Kenner: Fragmente zu **Unit-Tests in Ruby**

```
class XYZ
…      # Hier Ihre zu testenden Implementierungen
end

# Testcode in separater Datei, z.B. 02-ctests.rb
#
require "./02-container.rb"  # Einbinden des zu testenden Codes
require "test/unit"          # Test-Bibliothek einbinden

class TestXYZ < Test::Unit::TestCase # Eine Testklasse, erbt

  def test_feature_1         # Test-Methode (sinnvoll benennen!)
    assert_equal XYZ.new.class, XYZ
  end
  # … usw. (weitere Testmethoden. )
  test "feature 2" do        # Alternative Syntax, BDD-Stil
    assert XYZ.new.methods.size > 0 # nicht sehr sinnvoll, nur Demo
  end
end
```

LV 7281
Skriptsprachen-Praktikum
Übung 02

Python-Klassen und Methoden anlegen

Die Container-Klasse **list**

Organisatorisches

- Arbeitsverzeichnis:

  ```
  ~/lv/skriptspr/02/
  ```

- Dateinamen:

  ```
  p02_container.py # neu erstellen & abgeben
  p02_ctests.py    # neu erstellen & abgeben
  ```

- Werkzeuge:

  ```
  python           # Der Interpreter
  ```

- Vorlagen:

  ```
  (keine)
  ```

Die Aufgabe

- Allgemeine Beschreibung
  - Erzeugen Sie zwei „Klassiker“ der Informatik als einfache Klassen auf der Basis der eingebauten Python-Klasse „List”
  - Entwickeln Sie dazu passenden Testcode

- Material, Hinweise (Python):
  - Nutzen Sie Attribute zum Speichern der internen Werte ihrer Objekte.
  - Nutzen Sie das Delegationsprinzip – delegieren Sie die eigentliche Arbeit an Methoden der Klasse `list`
  - Informationen, auch zu verfügbaren List-Methoden, finden Sie hier:
    - https://www.w3schools.com/python/python_lists.asp

Die Aufgabe

A: Implementieren Sie eine Klasse **MyQueue** (FIFO) mit folgenden Methoden:

```
__init__(self, max_size=0)
  # Eine leere Warteschlange mit max_size Plätzen
  #  anlegen. max_size==0 : Keine Platzbegrenzung

enqueue(self, obj [, obj2, …])  --> anInt
  # Objekt hinten in Warteschlange einreihen
  # Rückgabewert = Anzahl übernommener Objekte
  #    Normalfall: 0 (kein Platz mehr) oder 1
dequeue(self)                   --> anObject
  # Objekt am vorderen Ende der Warteschlange entnehmen
  # und zurückliefern; None falls Queue leer
peek(self)                      --> anObject
  # Referenz auf Objekt am Ausgang (vorderen Ende) der
  # Warteschlange liefern, None falls Queue leer
```

Die Aufgabe

B: Implementieren Sie eine Klasse **MyStack** (LIFO) mit folgenden Methoden:

```
__init__(self, max_size=0)
  # Einen leeren Stack mit max_size Fächern anlegen
  #  max_size==0 : Keine Platzbegrenzung

push(self, obj [, obj2, …]) --> anInt
  # Objekt auf den Kellerspeicher legen
  # Rückgabewert = Anzahl übernommener Objekte
  #    Normalfall: 0 (kein Platz mehr) oder 1
pop(self)                   --> anObject
  # Objekt vom Kellerspeicher nehmen und zurückgeben
  # None falls Stack leer
peek(self)                  --> anObject
  # Referenz auf oberstes Objekt des Stacks liefern,
  # None falls Stack leer
depth (self)                --> anInt
  # Anzahl gespeicherter Objekte.
```

Die Aufgabe

C: Implementieren Sie die folgenden gemeinsame Methoden der Klassen **MyQueue** und **MyStack**:

```
length(self), size(self)        --> anInt
  # Anzahl gespeicherter Objekte.

clear(self)                     --> anInteger
  # Queue bzw. Stack löschen,
  # Anzahl Einträge vor dem Löschen zurückgeben.

empty(self)                     --> aBoolean
  # True, falls Queue bzw. Stack leer
```

Die Aufgabe

C: Schreiben Sie Test-Code zu beiden Klassen

Testen Sie möglichst viele Aspekte der Klassen

a) `Testen Sie jede Methode.`
b) `Testen Sie insbesondere Grenzfälle wie leere Warteschlange oder Stack-Überlauf.`
c) `Ausgabe einer Test-Statistik?`

HINWEISE

- Der Nutz-Code soll in der Datei „p02_container.py“ stehen.
- Der Test-Code wird in die Datei „p02_ctests.py“ geschrieben.
- Einbinden mit **`import p02_container`** (ohne .py)
- Achtung: Der Dozent wird Ihren Nutz-Code mit seinen eigenen Unit-Tests automatisiert prüfen!

(*) Ausbaustufe 1: Redundanzen vermeiden mittels einer gemeinsamen Basisklasse „Container“

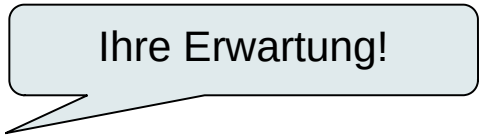
(*) Ausbaustufe 2: enqueue() und push() akzeptieren auch eine Liste von Objekten.

Bem.: (*) heißt hier: Optionaler Aufgabenteil

Kleine Hilfe

Hilfestellung: Etwas Source-Code als Ausgangspunkt

```
# Nutz-Code in p02_container.py:

class MyQueue:
    def __init__(self,max_size=0):
        self._q = []
        self._maxSize = max_size

    def size(self):
        return len(self._q)   # Arbeit an _q delegieren…


# Test-Code in p02_ctests.py:

import p02_container

q = p02_container.MyQueue()
if q.size() > 0:
    print("Fehler 1")
```



Kleine Hilfe

Für **JUnit**-Kenner: Fragmente zu **Unit-Tests in Python**

```
class XYZ:
    …      # Hier Ihre zu testenden Implementierungen

# Testcode in separater Datei, z.B. p02_ctests.py
#
import unittest          # Test-Bibliothek einbinden
import p02_container     # Einbinden des zu testenden Codes


class TestXYZ(unittest.TestCase):    # Eine Testklasse, erbt
    def test_feature_1(self):        # Test-Methode (sinnvoll benennen!)
        x = p02_container.XYZ()
        self.assertEqual(0, x.someMethod())

    # … usw. (weitere Testmethoden. )
    def test_feature_2(self):
        assertTrue(XYZ().__class__ == p02_container.XYZ) # nur Demo
```