Laboratorio 9

Elementi di Informatica e Programmazione

Esercizio 1: code e pile

Scaricate il codice `linked_list.py` da moodle che contiene anche le implementazioni delle classi `Stack` e `Queue`.

Ricordate che `Stack` e `Queue`:

- hanno entrambe un metodo `push` e un metodo `pop`
- per la classe `Stack` il metodo `push` aggiunge un elemento in fondo alla lista, il metodo `pop` rimuove e restituisce l'elemento in fondo alla lista (strategia LIFO: Last In First Out).
- per la classe `Queue` il metodo `push` aggiunge un elemento in fondo alla lista, il metodo `pop` rimuove e restituisce l'elemento in cima alla lista (strategia FIFO: First In First Out).

Modificare il codice in modo che: - il metodo `pop` invocato su uno `Stack` vuoto sollevi l'eccezione `EmptyStackError` - il metodo `pop` invocato su una `Queue` vuota sollevi l'eccezione `EmptyQueueError` - entrambe le classi abbiano un metodo `is_empty` che restituisce `True` se e solo se non ci sono elementi contenuti nella struttura dati

Le classi di entrambe le eccezioni devono essere create in maniera appropriata.

Soluzione

```
# Codice di LinkedList omesso per brevità

class EmptyStackError(Exception):
    pass

class EmptyQueueError(Exception):
    pass
```

```
class Stack:
    def __init__(self):
        self._list = LinkedList()

    def push(self, element):
        self._list.push_front(element)

    def pop(self):
        if self._list.is_empty():
            raise EmptyStackError()
        return self._list.pop_front()

    def is_empty(self):
        return self._list.is_empty()

class Queue:
    def __init__(self):
        self._list = LinkedList()

    def push(self, element):
        self._list.push_back(element)

    def pop(self):
        if self._list.is_empty():
            raise EmptyQueueError()
        return self._list.pop_front()

    def is_empty(self):
        return self._list.is_empty()
```

Esercizio 2: bilanciamento parentesi

Scrivere il programma `checkMatchingBrackets.py` che verifichi la corretta corrispondenza di parentesi chiuse e aperte (tonde, quadre e graffe) all'interno di un testo letto da un file interropendosi al primo errore individuato e fornendo una segnalazione d'errore che sia opportunamente informativa.

Suggerimento: usate la classe `Stack` sviluppata nell'esercizio precedente

Soluzione

```
from linked_list import Stack, EmptyStackError

def check_balanced(text):
    open_paren = "{(["
    close_paren = "})]"
    matching = {
        "}": "{",
        "]": "[",
        ")": "("
    }
    paren_stack = Stack()
    for character in text:
        if character in open_paren:
            paren_stack.push(character)
        elif character in close_paren:
            try:
                last_open = paren_stack.pop()
            except EmptyStackError:
                print("Parentesi", character, "chiusa senza essere stata precedentemente a
                return False
            if matching[character] != last_open:
                print("La parentesi", character,
                      "non può chiudere la parentesi", last_open,
                      "precedentemente aperta")
                return False
    if not paren_stack.is_empty():
        print("Alcune parentesi non sono state chiuse")
        return False
    return True

def check_file(path):
    with open(path, "r", encoding="utf-8") as fh:
        text = fh.read()
        return check_balanced(text)

def main():
    import sys
    # Leggiamo il nome del file come primo parametro
    path = sys.argv[1]
    if not check_file(path):
```

```
        print("Il file contiene errori nel bilanciamento delle parentesi")
```

Esercizio 3: Memory cell

Nel file `memoryCell.py`, progettare un modulo contenente la classe MemoryCell, che rappresenti una cella di memoria capace di memorizzare un valore di tipo int:

```
class MemoryCell :
    def __init__(self, v) :
        self._val = v

    def getVal(self) :
        return self._val

    def setVal(self, newVal) :
        self._val = newVal

    def clear(self) :
        self.setVal(0)
```

Aggiungere al modulo la classe `BackupMemoryCell`, sottoclasse di `MemoryCell`, che sia in grado, quando venga invocato il suo metodo restore privo di parametri, di ripristinare il valore immediatamente precedente all'ultima operazione di variazione del valore contenuto al suo interno.

Due invocazioni consecutive di restore, senza che venga modificato il contenuto della cella in altro modo, sono da considerarsi una violazione di stato, segnalata sollevando l'eccezione `IllegalStateError`, da progettare.

Aggiungere al modulo opportuno codice di collaudo per entrambe le classi.

Soluzione

```
class IllegalStateError(BaseException) :
    pass

class BackupMemoryCell(MemoryCell) :

    def __init__(self, v) :
        super().__init__(v)
```

```
        self._restoreIsAllowed = False
        self._backupValue = None

    def setVal(self, newVal) :
        self._backupValue = self.getVal()
        super().setVal(newVal)
        self._restoreIsAllowed = True

    def restore(self) :
        if not self._restoreIsAllowed :
            raise IllegalStateError
        super().setVal(self._backupValue)
        self._restoreIsAllowed = False

def main():
    def error() : print("ERRORE")
    m = BackupMemoryCell(2)
    if m.getVal() != 2 : error()
    m.setVal(3)
    if m.getVal() != 3 : error()
    m.restore()
    if m.getVal() != 2 : error()
    try :
        m.restore()
        error() # non deve arrivare qui, deve sollevare eccezione...
    except IllegalStateError :
        pass # OK
    m.setVal(4)
    if m.getVal() != 4 : error()
    m.restore()
```