

Lab 3

Instructions:

1. Provide your solutions in a file named **lab3.py**. In a comment, include the name(s) of any collaborators.
2. To receive full credit, the **names** of files, functions and the **output** must be **exactly** as indicated here.
3. **Test your code** by downloading the file **lab3TEST.py** in the same working folder and including at the bottom of your file:

```
if __name__ == '__main__':  
    import doctest  
    print( doctest.testfile('lab3TEST.py') )
```

Problems:

1. (based on 8.38) A stack is a sequence container type that, like a queue, supports very restrictive access methods: all insertions and removals are from one end of the stack, typically referred to as the top of the stack. A stack is often referred to as a list-in first-out (LIFO) container because the last item inserted is the first removed. Implement a `Stack` class. It should support the following methods/functions:
 - a. `__init__` - Can construct either an empty stack, or initialized with a list of items, the first item is at the bottom, the last is at the top.
 - b. `push()` - take an item as input and push it on the top of the stack
 - c. `pop()` - remove and return the item at the top of the stack
 - d. `isEmpty()` - returns `True` if the stack is empty, `False` otherwise
 - e. `[]` - return the item at a given location, `[0]` is at the bottom of the stack
 - f. `len()` - return length of the stack

The object is to make this client code work:

```
'''  
>>> s = Stack()  
>>> s.push('apple')  
>>> s  
Stack(['apple'])
```

```

>>> s.push('pear')
>>> s.push('kiwi')
>>> s
Stack(['apple', 'pear', 'kiwi'])
>>> top = s.pop()
>>> top
'kiwi'
>>> s
Stack(['apple', 'pear'])
>>> len(s)
2
>>> s.isEmpty()
False
>>> s.pop()
'pear'
>>> s.pop()
'apple'
>>> s.isEmpty()
True
>>> s = Stack(['apple', 'pear', 'kiwi'])
>>> s = Stack(['apple', 'pear', 'kiwi'])
>>> s[0]
'apple'
>>>
'''

```

2. Write a client function `parenthesesMatch` that given a string containing only the characters for parentheses, braces or curly braces, i.e., the characters in `'([{}])'`, returns `True` if the parentheses, brackets and braces match and `False` otherwise. Your solution must use a `Stack`. For, example:

```

>>> parenthesesMatch('(){}[]')
True
>>> parenthesesMatch('{[()]}' )
True
>>> parenthesesMatch('((())){[()]}' )
True
>>> parenthesesMatch('()')
False
>>> parenthesesMatch('({})')
False
>>> parenthesesMatch('((()'))' )
False
>>> parenthesesMatch('((()))')
False

```

>>> Hint: It is not sufficient to just count the number of opening and closing marks. But, it is easy to write this as a simple application of the `Stack` class. Here is an algorithm:

1. Create an empty stack.
2. Iterate over the characters in the given string:
 - a. If the character is one of opening marks `(, [, {` push it on the stack.
 - b. If the character is one of the closing marks `),], }` and the stack is empty, then there were not enough preceding opening marks, so return `False`.
 - c. If the character is a closing mark and the stack is not empty, pop an (opening) mark from the stack. If they are not of the same type, ie., `(` and `)` or `[` and `]` or `{` and `}`, return `False`, if they are of the same type, move on to the next char.
3. Once the iteration is finished, you know that the parentheses match if and only if the stack is empty.