## HW8

### **Reading**: Chapter 8

### **Programming**:

Submit a single file named hw8.py that contains the solutions to the two problems below. When you are finished, test your solutions using doctest. Include the following code at the bottom of your module in order to run the doctest:

if \_\_name\_\_=='\_\_main\_\_':

import doctest

print( doctest.testfile( 'hw8TEST.py'))

1. (Composition) Write a Pizza class so that this client code works. Please note that it is ok if the toppings are listed in a different order.

>>> pie = Pizza()

>>> pie

Pizza('M',set())

>>> pie.setSize('L')

>>> pie.getSize()

'L'

>>> pie.addTopping('pepperoni')

>>> pie.addTopping('anchovies')

>>> pie.addTopping('mushrooms')

>>> pie

Pizza('L',{'anchovies', 'mushrooms', 'pepperoni'})

>>> pie.addTopping('pepperoni')

>>> pie

Pizza('L',{'anchovies', 'mushrooms', 'pepperoni'})

>>> pie.removeTopping('anchovies')

>>> pie

Pizza('L',{'mushrooms', 'pepperoni'})

>>> pie.price()

16.65

>>> pie2 = Pizza('L',{'mushrooms','pepperoni'})

>>> pie2

Pizza('L',{'mushrooms', 'pepperoni'})

>>> pie==pie2

True

The Pizza class should have two attributes(data items):

size – a single character str, one of ‘S’,’M’,L”

toppings – a set containing the toppings. If you don’t remember how to use a set, make sure you look it up in the book. Please note that toppings may be listed in a different order, but hw2TEST.py takes that into account.

The Pizza class should have the following methods/operators):

\_\_init\_\_ **-** constructs a Pizza of a given size (defaults to ‘M’) and with a given set of toppings (defaults to empty set). I highly recommend you look at the Queue class in the book to see how to get this to work correctly.

setSize – set pizza size to one of ‘S’,’M’or ‘L’

getSize – returns size

addTopping – adds a topping to the pizza, no duplicates, i.e., adding ‘pepperoni’ twice only adds it once

removeTopping – removes a topping from the pizza

price – returns the price of the pizza according to the following scheme:

‘S’: $6.25 plus 70 cents per topping  
 ‘M’: $9.95 plus $1.45 per topping  
 ‘L’: $12.95 plus $1.85 per topping

\_\_repr\_\_ - returns representation as a string – see output sample above. Note that toppings may be listed in a different order.

\_\_eq\_\_ - two pizzas are equal if they have the same size and same toppings (toppings don’t need to be in the same order)

1. Write a function orderPizza that allows the user input to build a pizza. It then prints a thank you message, the cost of the pizza and then **returns** the Pizza that was built.

>>> orderPizza()

Welcome to Python Pizza!

What size pizza would you like (S,M,L): M

Type topping to add (or Enter to quit): mushroom

Type topping to add (or Enter to quit): onion

Type topping to add (or Enter to quit): garlic

Type topping to add (or Enter to quit):

Thanks for ordering!

Your pizza costs $14.299999999999999

Pizza('M',{'mushroom', 'onion', 'garlic'})

>>> orderPizza()

Welcome to Python Pizza!

What size pizza would you like (S,M,L): L

Type topping to add (or Enter to quit): calamari

Type topping to add (or Enter to quit): garlic

Type topping to add (or Enter to quit):

Thanks for ordering!

Your pizza costs $16.65

Pizza('L',{'garlic', 'calamari'})

>>> p=orderPizza()

Welcome to Python Pizza!

What size pizza would you like (S,M,L): S

Type topping to add (or Enter to quit):

Thanks for ordering!

Your pizza costs $6.25

>>> p

Pizza('S',set())

>>>

1. Inheritance (based on 8.38) You MUST use inheritance for this problem. 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 last-in first-out (LIFO) container because the last item inserted is the first removed. Implement a Stack class using **inheritance.** Note that this means you may be able to inherit some of the methods below. Which ones? (Try not writing those and see if it works!)
   1. Constructor/\_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.
   2. push() – take an item as input and push it on the top of the stack
   3. pop() – remove and return the item at the top of the stack
   4. isEmpty() – returns True if the stack is empty, False otherwise
   5. [] – return the item at a given location, [0] is at the bottom of the stack
   6. 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'

>>>

'''

1. 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(')(][') # right number, but out of order

False

>>> parenthesesMatch('([)]') # right number, but out of order

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:
   1. If the character is one of opening marks(,[,{ push it on the stack.
   2. 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.
   3. 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.